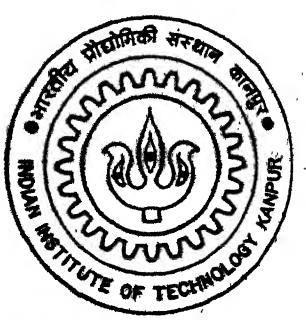
INCREMENTAL DATA DEPENDENCE ANALYSIS

by

PRAVEEN K. V.



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

APRIL, 1996

Incremental Data Dependence Analysis

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

Praveen K.V

to the

DEPARTMENT OF

COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, KANPUR

April, 1996.

CSE-1996-M-PRA-INC

Acknowledgments

First of all I wish to thank my famuly for encouraging me to do M. Tech and their constant support. I would like to thank Dr. Sanjeev Aggarwal & Dr. R. K. Ghosh for encouraging me to try out something new and interesting. Thier constant encourage and confidence has been of immence help. My friends in *IIT/K* deserve a special mention, they have been excellent company and we've had lots of fun together. And last, but certainly not the least, I wish to thank all the members of the "Kannada Sangha" especially Dr. Ragahavendra and Shantakka for making the stay memoriable.

Abstract

Automatic restructurers do not completely extract all the parallelism in sequential programs. As a result, Interactive parallelizers are considered an attractive alternative. In Interactive tools, the dependence updates in response to changes should be quick. However, computing value based dependences is a time consuming procedure. Changes in a program typically affect a small portion of dependences and leave larger part unaffected. The resulting dependences can be got faster than exhaustive analysis if only the affected portion is recomputed. Incremental analyzers modify only the affected parts of the solution, and hence can quickly reflect the affect of the changes on the solutions. In this work we present a framework for incremental dependence analysis for value based dependences.

We illustrate how the dependences change as a result of increase or decrease in the values generated at any point in a program. Using these techniques, we show how the incremental analysis can be done in cases of insertion or deletion of assignment statements and loops. We give algorithms for incremental analysis for these cases. We will also present the speedups obtained by implementations based on the algorithms given.

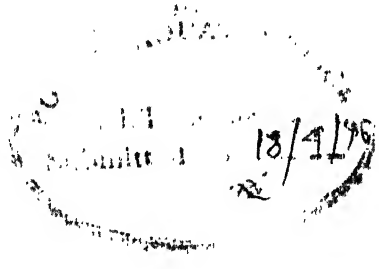
CERTIFICATE

This is to certify that the work contained in the thesis entitled *" Incremental Data Dependenece Analysis "* by *"Praveen K.V"* (Roll No: 9411126), has been carried out under our supervision and that this work has not been submitted elsewhere for a degree.

Dr. Sanjeev Kumar Aggarwal
Associate Professor
Department of Computer
Science & Engineering,
Indian Institute of Technology,
Kanpur.

Dr. R.K. Ghosh
Associate Professor
Department of Computer
Science & Engineering,
Indian Institute of Technology,
Kanpur.

April, 1996



Contents

List of Figures

Chapter 1

Introduction

Von Neumann paradigm of computing advocated sequential semantic of execution of programs. As a result, all the programs written in earlier days used only sequential constructs. The growing need for computing power and the progress in technology resulted in a shift in computing paradigm and development of Vector and Parallel computers. However, the serial code accumulated over the years cannot exploit the power of these computers.

Re-coding all sequential code using parallel constructs in not economical. Smart compilers that can identify parallelism in the programs written with sequential constructs were developed. These are called "Parallelizing Compilers" or "Restructuring Compilers". These compilers re-order the execution sequence of the statements to execute some of them in parallel.

Data dependence analysis is an important procedure in identifying parallelism in a sequential code. It identifies the flow of values between the statements. This information is used to check the feasibility of the re-ordering transformations, such that the semantics are maintained. However, computing dependences is a costly process. It is known to be NP-Complete [11].

The existing restructuring compilers cannot extract all the parallelism in a sequential program. As a result, interactive parallelizing tools are considered an attractive alternative. These tools can be of multipurpose use and can be used for interactive optimizing of the parallel programs also. In such tools, often the users

make small modifications to the programs. Under the existing framework for dependence analysis, every time a program is modified, exhaustive reanalysis has to be carried out to restructure the program. Often the changes in the program may be limited to a small portion, and may not affect a major part of the value based dependences. Therefore, modifying only the affected part of the dependences can result in a faster computation of the dependences.

1.1 The Problem

The proposed problem to be tackled is,

- to identify the dependences affected as a result of modifications to the program and
- to compute the new dependences, as well as update the affected dependences to reflect the effect of the modifications."

1.2 Work Done

In this work, we have proposed a framework for incremental updating of value based dependences. We show how the dependences are affected on arbitrary increase/decrease in the values generated and the values read. Using these techniques, we have developed algorithms for insertion/deletion of assignment statements and loops. However, the techniques are general enough to be extended for other form of modifications also.

1.3 Related Work

Smith, Applebe and Stirewalt [4] have developed a program assistance tool (PAT) for incrementally updating of memory based dependence information. However, in their method the global information remains static as sequential dependence graphs are not modified. This restricts the incremental updating only to specific locations of the code.

1.4 Organization

The organization of the report is as follows :

Chapter 2 Brief introduction to the basic concepts of value based dependence and its representations.

Chapter 3 Discusses the incremental analysis of dependences for increase and decrease in the values generated. The incremental analysis for insertion/deletion of assignment statements and for loops is also discussed in this chapter.

Chapter 4 Gives the algorithm implemented.

Chapter 5 Gives the results of implementation.

Chapter 6 Gives the conclusions and discusses the future scope.

Appendix A Gives the exhaustive dependence analysis algorithm, "Lazy Array Data-Flow Dependence Analysis".

Chapter 2

Basic Concepts

In this chapter we briefly introduce some of the basic concepts of dependence and the representations used for dependence. The notations used in this work are given in figure 1.

2.1 Dependence

There is a data dependence from reference $S_i.x$ to statement $S_j.y$ if in some execution sequence, there exist $S_i[i_1, s]$ and $S_j[i_2, s]$, instances of S_1 and S_2 respectively such that

- both $S_i.x[i_1, s]$ and $S_j.y[i_2, s]$ access a location, say M
- $S_i.x[i_1, s]$ or $S_j.y[i_2, s]$ is a write to a location M
- $S_1[i_1, s]$ is executed before $S_j[i_2, s]$

If there is a dependence from $S_i.x$ to $S_j.y$, $S_i.x$ is called source of the dependence and $S_j.y$ is called sink of the dependence. Depending on whether $S_i.x$ and $S_j.y$ is read or write, Kuck and others[5] classified dependences as :

Flow or True Dependence : In this dependence, $S_i.x$ is a write to the location M and $S_j.y$ is a read from M. Flow dependence is denoted by δ and dependence between statements is written as $S_i.x\ \delta\ S_j.y$.

	The Entry of the program
i	Specific statements in a program
	The universal set - consisting all assignment statements.
$rr(S_i.k)$	Array or the scalar variable referred in reference $S_i.k$.
$, r$	The iteration vectors that represent specific set of values of loop variables.
$[i \ldots j]$	The sub vector of w consisting of components $w_i, w_{i+1}, \ldots, w_j$.
$_i.k$	Specific references in the statement S_i, in order from left to right. The write in an assignment is always $S_i.1$.
$S_i, s]$	The set of iteration vectors for which statement S_i is executed, given set of symbolic constants s.
$_i[i_1, s]$	Specific instance of statement S_i executed for iteration vector i_1.
$\ll$	defines the order on the two operands. eg., $S_i \ll S_j$ implies S_i occurs before S_j lexically, $S_i[i_1, s] \ll S_j[i_2, s]$ implies statement instance $S_i[i_1, s]$ is executed before the instance $S_j[i_2, s]$.
$)epRels_{j.x}$	The set of dependence relations for reference $S_j.x$, such that $S_j.x$ is the sink.
$)epRels_{S_i S_j.x}$	Dependence relation representing dependence from $S_i.1$ to $S_j.x$
$)epRels_{S_i S_j.k} \setminus V$	Restrict the domain of relation $DepRels_{S_i S_j.k}$ to the vector V. (x → y) $\epsilon DepRels_{S_i S_j.k} \setminus S \Leftrightarrow$ (x → y) $\epsilon DepRels_{S_i S_j.k} \wedge x \epsilon S$
$DepRels_{S_i S_j.k} / V$	Restrict the range of relation $DepRels_{S_i S_j.k}$ to the vector V. (x → y) $\epsilon DepRels_{S_i S_j.k} / S \Leftrightarrow$ (x → y) $\epsilon DepRels_{S_i S_j.k} \wedge y \epsilon S$

Figure 1: Notations

```
                For i1 = 1, 5
                    For i2 = 1, 4
S1 :                    A(i1, i2) = B(i1, i2)
S2 :                    B(i1, i2 + 1) = A(i1, i2) + B(i1, i2)
S3 :                    A(i1, i2 + 1) = 2
                    Endfor
                Endfor
```

Figure 2: Example code

Output Dependence : In this dependence, both $S_i.x$ and $S_j.y$ are write to the location M. Output dependence is denoted by δ^o and the dependence between the statements is written as $S_i.x\ \delta^o\ S_j.y$.

Anti Dependence : In this dependence, $S_i.x$ is a read from the location M and $S_j.y$ is a write to location M. The dependence is denoted by $\bar{\delta}$ and the dependence between the statements is written as $S_i.x\bar{\delta}S_j.y$.

δ^* represents dependence without specifying the type.

In figure 2, the reference $A(i_1, i_2)$ in statement S_2 reads from the memory locations that are written in statement S_1, in the same iterations of i_1 and i_2. So there is a flow dependences from S_1 to S_2.

The reference $B(i_1, i_2 + 1)$ in statement S_2 writes to the location that was read by reference $B(i_1, i_2)$ in S_2 in the previous iterations of i_2. So there is an anti dependence from S_2 to S_1.

The reference $A(i_1, i_2)$ in the statement S_3 writes to the location written by $A(i_1, i_2 + 1)$ in the previous iteration of i_2. So there is an output dependence from S_3 to S_1.

The output and anti dependences are a result of reuse of the memory locations. They can be removed by different transformations [9]. Flow dependence is a result of reuse of the values defined at the source by the sink. For this reason flow dependence is also called as true dependence.

2.2 Representations

The dependences are traditionally represented by *distance vectors* and *direction vectors*.

2.2.1 Distance Vector

The distance vector [9] for a dependence $S_i.x[w,s] \rightarrow S_j.y[r,s]$, is an n-tuple $D = (D_1, D_2, \ldots, D_n)$ where,

- S_i and S_j are covered by n loops and
- $D_i = w_i - r_i$.

2.2.2 Direction Vector

The direction vector [6] for a dependence $S_i.x[w,s] \rightarrow S_j.y[r,s]$, is an n-tuple $d = (d_1, d_2, \ldots, s_n)$ where,

- S_i and S_j are covered by n loops and
- $d_i \in \{<, \leq, >, \geq, =, *\}$ such that the logical relation $w_i\ d_i\ r_i$ holds.

Direction vectors are more coarse form of representing dependence than distance vectors. The direction vector d can be obtained from distance vector D, where

$$d_i = \begin{cases} `>' \ if\ D_i > 0 \\ `=' \ if\ D_i = 0 \\ `<' \ if\ D_i < 0 \\ `\leq' \ if\ D_i \leq 0 \\ `\geq' \ if\ D_i \geq 0 \\ `*' \ if\ any\ of\ the\ above\ relations \end{cases}$$

2.3 Loop Carried and Loop Independent dependence

Dependences can also be classified as "*Loop Carried*" dependence or "*Loop Independent*" dependence.

If a dependence is a *Loop Independent* dependence all the elements in the direction vector are '='. If a dependence is loop independent dependence the last common loop between the source and sink say, l_n is called as *carrier* of the dependence.

If a dependence is a *Loop Carried* dependence at least one element in the direction vector is not '='. A loop l_i is called *carrier* of the loop carried dependence if, the first non '=' element in the direction vector is '$i + 1^{th}$' element.

2.4 Restricted Domain For Dependence Calculation

The problem of finding dependence in general is undecidable. In practice it is enough to consider dependences that are affine, as most of the references are of this category. The dependence problem is affine [8] if,

- All the loop bounds are integers or integer linear functions of more outwardly nested loop variables and
- The array reference functions are integer linear functions of the loop variables, or symbolic constants.

Most of the analyzers ignore control flow in calculating the dependences [8], or restrict the conditions in the control statements also to be affine. In this restricted domain, the problem of finding dependence is NP-complete [11].

2.5 Value Based Dependence

Dependences define constraints on the order of execution of the statements. After restructuring, the resulting program must maintain these constraints. The definition

```
1        For i = 1, 5
S1 :         A(i) = ...
S2 :         A(i) = A(i - 1)
         Endfor
```

(a)

```
                    For i = 1, 5
S1:                     A(i) = ...
                    Endfor
                    For i = 1, 5
S2:                     A(i) = A(i - 1)
                    Endfor
```

(b)

Figure 3: Example code

above say there is a dependence between two references if they access same memory location. This definition is too weak and introduces false dependences which may not allow transformations that could have been applied otherwise. According to the definitions given above the dependences in the program given in figure 3 are,

- Output dependence from $A(i)$ in S_1 to $A(i)$ in S_2.
- Output dependence from $A(i)$ in S_2 to $A(i)$ in S_1.
- Flow dependence from $A(i-1)$ in S_2 to $A(i)$ in statement S_1.

Though the program in figure 3 can be transformed to the program in figure 3(b) the flow dependence from S_1 to S_2 does not allow the transformation. This is because the restructuring is restricted by the flow of values between the references rather than reference to the same memory location as assumed by the definition above. It is called *Memory Based Dependence.*

Feautrier [3] restricted the definition of dependence to represent the true flow of values, and called this form of dependence as *Value Based Dependence.* According to his definition there is a value based dependence from reference $S_i.x$ to reference $S_j.y$ if there are $S_i[i_1, s]$ and $S_j[i_2, s]$, instances of S_i and S_j respectively, such that

- at least one of the references $S_i.x$ or $S_j.y$ is write
- statement instance $S_i[i_1, s]$ is executed before the statement instance $S_j[i_2, s]$
- the references $S_i.x$ and $S_j.y$ refer to the same memory location M in instances $S_i.x[i_1, s]$ and $S_j.y[i_2, s]$ and

- there is no overwriting of value in the memory location M between the instances $S_i[i_1, s]$ and $S_j[i_2, s]$.

According to the above definition the value based dependences in the program given in figure 3 are

- Output dependence from $A(i)$ in S_1 to $A(i)$ in S_2.
- Flow dependence from $A(i-1)$ in S_2 to $A(i)$ in S_2.

Now the program in figure 3 can be transformed to the program given in figure 3(b).

2.6 Dependence Relations

The direction vectors and the distance vectors are powerful enough to represent the memory based dependences and the transformations used traditionally. However, value based dependences as well as some of new transformations such as array expansion [2] and array privatization [1] need more powerful representations. In these transformations, given a reference and a set of iterations, it should be possible to calculate the source and its iterations involved in dependence. Neither of the above representations can identify the exact iterations involved for all the cases. The direction vectors do not identify the exact iterations of dependence except for direction '='. The distance vectors can identify the iterations only if the distance between the source and sink iterations is constant. Recent research on value based dependence and the representations, resulted in *Quasi-Affine Selection Tree(quast)* [3] also called *Last Write Tree* and *Dependence Relations* [10]. Given an iteration at the sink, the source and the instance of the source can be found in these functions. Hence, they are called as *source functions* [3].

Dependence relation is a form of source functions. It is a mapping from the iteration space of the sink to the iteration space of the source. The mapping is specified by a set of linear constraints on loop indices, subscript functions and the symbolic constants at the source and the sink.

For example, in figure 3, the flow dependence from the read $A(i)$ in statement S_2 to the write $A(i)$ in S_2 is represented in dependence relations as,

$$S_2.1[i_1] \rightarrow S_2.2[i_2] \quad | \quad 2 \leq i_2 \leq 5 \ \wedge \ i_1 = i_2 - 1 \tag{1}$$

In this report, the dependences are represented using dependence relations. A dependence relation consists of two parts, viz.,

1. The reference involved in the dependence. For example $S_2.1[i_1] \rightarrow S_2.2[i_2]$ in equation 1 represents the references.

2. The conditions on the index variables. These conditions map a vector to a boolean value. If the conditions evaluate to *true* for a given vector, then the first reference given in (1) is the source. For example, $2 \leq i_2 \leq 5 \ \wedge \ i_1 = i_2 - 1$ in equation 1 gives the conditions. It means that for all the iterations i_2, for $i_2, 2 \leq i_2 \leq 5$ of the sink $S_2.2$ the source is $S_2.1$. Furthermore, the iteration at source is equal to the iteration i_2 at the sink.

In Dependence relations the loop indices for the source are expressed as functions of the loop indices at the sink. The affine restrictions given above allow the indices to be multiplied by integers. When these are converted to functional form, the corresponding linear constraint becomes *non-affine*. For example, if the condition on the index variable i is $c * i = k$, c being a constant, in functional form the constraint becomes $i = \lfloor k/c \rfloor$. This is not in affine form. For such cases, Pugh [10] proposed to use an additional wild-card variable and convert it to affine form. The above constraint can be written as as $c * i + \alpha = k \ \wedge \ 0 \leq \alpha < c$. However, for clarity, such constraints are shown in non-affine form in this work.

Chapter 3

Basis Of Incremental Analysis

In this chapter, we show how dependences change on increase/decrease of the values generated/read. Using these techniques, we show how the dependences can be updated for insertion/deletion of the assignment statements and for loops.

The changes in a program may result in increase/decrease in values generated or increase/decrease in values read or a combination of both. The value based dependences get affected due to increase or decrease in the values generated or read. In this work we have considered incremental updating of the flow and output dependences. This can be extended in a straight forward manner for anti dependence also. Lemmas 3.1, 3.2 and 3.3 given below, captures the concept of value based dependences and provide basis for the proofs given in this chapter.

Lemma 3.1 *If there is a dependence* $S_i.1[i_1, s] \rightarrow S_j.x[i_2, s]$, *and* $S_j.x[i_2, s]$ *access the location M, then* $S_i.1[i_1, s]$ *is the last write to M before* $S_j.x[i_2, s]$ *is executed.*∎

Lemma 3.2 *The Entry E, defines value for all the memory locations used in a program. These are the first values defined for the memory locations.*

Lemma 3.3 *The values generated by E remain unaltered by any modification to the program.*

3.0.1 Increase In Values Generated

The increase in the values generated introduces dependences involving the new values. This may also invalidate some of the old dependences. Theorem 3.6 and 3.7 given below describe how the dependences change on increase of values generated. Lemma 3.4 and 3.5 which follows, provide the basis of the proofs for the theorems.

Lemma 3.4 *Let there be a dependence* $S_i.1[i_1, s] \rightarrow S_j.1[i_2, s]$. *Suppose a dependence* $S_i.1[i_1, s] \rightarrow S_k.1[i_3, s]$ *appears due to an increase in the generated values then* $S_i.1[i_1, s] \ll S_k.1[i_3, s] \ll S_j.1[i_2, s]$.

Proof : Let $S_i.1[i_1, s]$ write to the memory location M then $S_j.1[i_2, s]$ and $S_k.1[i_3, s]$ also write to M. The dependence from $S_i.1[i_1, s]$ to $S_j.1[i_2, s]$ implies, $S_i.1[i_1, s] \ll S_j.1[i_2, s]$ and the dependence from $S_i.1[i_1, s]$ to $S_k.1[i_3, s]$ implies, $S_i[i_1, s] \ll S_k[i_3, s]$ (by definition of dependences). If $S_j[i_2, s] \ll S_k[i_3, s]$ then the source for $S_k.1[i_3, s]$ is $S_j.1[i_2, s]$, because both of them write to the same memory location M, and $S_i[i_1, s] \ll S_j[i_2, s]$. But the dependence from $S_i.1[i_1, s]$ to $S_k.1[i_3, s]$ contradicts this. Hence $S_k.1[i_3, s] \ll S_j.1[i_2, s]$. The dependence $S_i.1[i_1, s] \rightarrow S_k.1[i_3, s]$ implies $S_i.1[i_1, s] \ll S_k.1[i_3, s]$. Hence, $S_i.1[i_1, s] \ll S_k.1[i_3, s] \ll S_j.1[i_2, s]$.∎

Lemma 3.5 *Let there be a dependence* $S_i.1[i_1, s] \rightarrow S_j.1[i_2, s]$. *Suppose a new dependence* $S_i.1[i_1, s] \rightarrow S_k.1[i_3, s]$, *is formed as a result of increase in the generated values then* $S_k.1[i_3, s]$ *is the last write before* $S_j.1[i_2, s]$.

Proof : Let $S_i.1[i_1, s]$ write to the memory location M then $S_k.1[i_3, s]$ and $S_j.1[i_2, s]$ also write to M. Assume that there is a write say $S_p.1[i_n, s]$ between $S_k.1[i_3, s]$ and $S_j.1[i_2, s]$. By Lemma 3.4, $S_i.1[i_1, s] \ll S_k.1[i_3, s] \ll S_j.1[i_2, s]$. This implies that $S_i.1[i_1, s] \ll S_p.1[i_n, s] \ll S_j.1[i_2, s]$. The dependence from $S_i.1[i_1, s]$ to $S_j.1[i_2, s]$ implies that there is no write between $S_i.1[i_1, s]$ and $S_j.1[i_2.s]$ (by Lemma 3.1). This contradicts the assumption above. Hence, $S_k.1[i_3, s]$ is the last write before $S_j.1[i_2, s]$ to the memory location M.∎

Theorem 3.6 *Let there be a dependence* $S_i.1[i_1, s] \rightarrow S_j.1[i_2, s]$. *If a new dependence* $S_i.1[i_1, s] \rightarrow S_k.1[i_3, s]$ *is formed due to increase in the generated values, then* $S_k.1[i_3, s] \rightarrow S_j.1[i_2, s]$.

Proof : Let $S_i.1[i_1, s]$ write to the memory location M. By Lemma 3.4, $S_k.1[i_3, s]$ is a write in between $S_i.1[i_1, s]$ and $S_j.1[i_2, s]$. By Lemma 3.5, $S_k.1[i_3, s]$ is the last write before $S_j.1[i_2, s]$ to the memory location M. By the definition of dependences it follows that there is a dependence from $S_k.1[i_3, s]$ to $S_j.1[i_2, s]$.∎

Theorem 3.7 *Let there be dependences of form* $S_i.1[i_1, s] \rightarrow S_j.x[i_2, s]$, *for* $x \neq 1$. *If a new dependence* $S_i.1[i_1, s] \rightarrow S_k.1[i_3, s]$ *is formed due to increase in the generated values, then* $S_k.1[i_3, s] \rightarrow S_j.x[i_2, s]$ *provided* $i_3 \ll i_2$.

Proof : Let $S_i.1[i_1, s]$ write to the memory location M, then $S_k.1[i_3, s]$ also writes to M. $S_j.x[i_2, s]$ writes to M if $x = 1$ or reads from M if $x \neq 1$. The dependences $S_i.1[i_1, s] \rightarrow S_j.x[i_2, s]$ and $S_i.1[i_1, s] \rightarrow S_k.1[i_3, s]$ implies $S_i.1[i_1, s] \ll S_j.x[i_2, s]$ and $S_i.1[i_1, s] \ll S_k.1[i_3, s]$.

Now, either $S_k.1[i_3, s] \ll S_j.x[i_2, s]$ or $S_j.x[i_2, s] \ll S_k.1[i_3, s]$. By Lemma 3.4, there is no write to M between $S_k.1[i_3, s]$ and $S_j.x[i_2, s]$.If $S_j.x[i_2, s] \ll S_k.1[i_3, s]$ then $S_j.x[i_2, s] \rightarrow S_k.1[i_3, s]$. If $S_k.1[i_3, s] \ll S_j.x[i_2, s]$ then $S_k.1[i_3, s] \rightarrow S_j.x[i_2, s]$. ∎

Example : Consider the program in the figure 4(a). The values generated by the statements are,

$$\begin{aligned} S_1[i] &\mid 1 \leq i \leq 3n \\ S_2[i] &\mid 1 \leq i \leq n \\ S_3[i] &\mid 1 \leq i \leq 3n \end{aligned}$$

The dependences in the program are,

$$\begin{aligned} S_1[i_1] \rightarrow S_2.1[i_2] &\mid 1 \leq i_2 \leq n \wedge i_1 = i_2 \\ S_2[i_1] \rightarrow S_3.1[i_2] &\mid 1 \leq i_2 \leq n \ \wedge \ i_1 = i_2 \\ S_1[i_1] \rightarrow S_3.1[i_2] &\mid n+1 \leq i_2 \leq 3n \ \wedge \ i_1 = i_2 \end{aligned}$$

Now if the upper limit of the loop l is increased from n to 2n as in figure 4(b), there is an increase in the values generated by the statement S_2. The new values generated are $S_2[i] \mid n+1 \leq i \leq 2n$. For these values, the source function is,

$$S_1[i_1] \rightarrow S_2.1[i_2] \mid n+1 \leq i_2 \leq 2n \ \wedge \ i_2 = i_1$$

	For $i = 1, 3n$	For $i = 1, 3n$
S_1 :	$A(i) = \ldots$	$A(i) = \ldots$
	Endfor	Endfor
l :	For $i = 1, n$	For $i = 1, 2n$
S_2 :	$A(i) = \ldots$	$A(i) = \ldots$
	Endfor	Endfor
	For $i = 1, 3n$	For $i = 1, 3n$
S_3 :	$A(i) = \ldots$	$A(i) = \ldots$
	Endfor	Endfor
	(a)	(b)

Figure 4: Changes in values generated

Before the increase of the loop limit, the source for $S_3.1$ for the values $S_3.1[i] \mid n+1 \leq i \leq 2n$ was S_1. After the change, the statement S_2 overwrites the values written by $S_1.1$ in the memory locations referred to by $S_3.1$ above. Hence, by Theorem 3.6 for the values above the source for $S_3.1$ is $S_2.1$. The new dependence in the program are,

$$S_1[i_1] \rightarrow S_2.1[i_2] \mid 1 \leq i_2 \leq 2n \wedge i_1 = i_2$$
$$S_2[i_1] \rightarrow S_3.1[i_2] \mid 1 \leq i_2 \leq 2n \wedge i_1 = i_2$$
$$S_1[i_1] \rightarrow S_3.1[i_2] \mid 2n+1 \leq i_2 \leq 3n \wedge i_1 = i_2$$

3.0.2 Increase in Values Read

Increase in values read will only generate new dependences involving these values. It does not affect any previous dependences. The incremental analysis for increase in values read is to compute the dependences above.

3.0.3 Decrease In Values Generated

Decrease in the values generated not only invalidates the dependences involving these values,but also generates new dependences as given in the Theorem 3.8 below.

Theorem 3.8 *Let there be dependences as* $S_k.1[i_3, s] \rightarrow S_i.1[i_1, s]$ *and* $S_i.1[i_1, s] \rightarrow S_j.x[i_2, s]$. *Suppose as a result of decrease in the values, if* $S_i.1[i_1, s]$ *is not generated then,* $S_k.1[i_3, s] \rightarrow S_j.x[i_2, s]$.

Proof : Let $S_i.1[i_1, s]$ write to the memory location M, then $S_k.1[i_3, s]$ also writes to M. $S_j.x[i_2, s]$ writes to M if $x = 1$ or reads from M if $x \neq 1$. The dependence $S_i.1[i_1, s] \rightarrow S_j.x[i_2, s]$ implies, $S_i.1[i_1, s] \ll S_j.x[i_2, s]$. The dependence $S_k.1[i_3, s] \rightarrow S_i.1[i_1, s]$ implies, $S_k.1[i_3, s] \ll S_i.1[i_1, s]$. Hence, $S_k.1[i_3, s] \ll S_j.x[i_2, s]$ Before the decrease in the values, $S_k.1[i_3, s]$ is the last write to M before $S_i.1[i_1, s]$ and $S_i.1[i_1, s]$ is the last write to M before $S_j.x[i_2, s]$. If $S_i.1[i_1, s]$ is not generated, then the last write to M before $S_j.x[i_2, s]$ is $S_k.1[i_3, s]$. Therefore, by the definition of value based dependence, $S_k.1[i_3, s] \rightarrow S_j.x[i_2, s]$. ∎

Example : Consider the program in figure 4(b). The values generated in the program are,

$$\begin{aligned} S_1.1[i] &\mid 1 \leq i \leq 3n \\ S_2.1[i] &\mid 1 \leq i \leq 2n \\ S_3.1[i] &\mid 1 \leq i \leq 3n \end{aligned}$$

The dependences in the program are,

$$\begin{aligned} S_1[i_1] \rightarrow S_2.1[i_2] &\mid 1 \leq i_2 \leq 2n \ \wedge \ i_1 = i_2 \\ S_2[i_1] \rightarrow S_3.1[i_2] &\mid 1 \leq i_2 \leq 2n \ \wedge \ i_1 = i_2 \\ S_1[i_1] \rightarrow S_3.1[i_2] &\mid 2n+1 \leq i_2 \leq 3n \ \wedge \ i_1 = i_2 \end{aligned}$$

If the upper limit of the index i is decreased from $2n$ to n, the resulting program is shown in figure 4(a). Now the statement S_2 does not generate the values, $S_2.1[i] \mid n+1 \leq i \leq 2n$. Statement S_1 is the immediately preceding write to the locations written by the values above, and S_3 is the immediately succeeding write to the locations after S_2. Hence, by Theorem 3.8, on decrease of values, the new dependences are,

$$\begin{aligned} S_1[i_1] \rightarrow S_2.1[i_2] &\mid 1 \leq i_2 \leq n \ \wedge \ i_1 = i_2 \\ S_2[i_1] \rightarrow S_3.1[i_2] &\mid 1 \leq i_2 \leq n \ \wedge \ i_1 = i_2 \\ S_1[i_1] \rightarrow S_3.1[i_2] &\mid n+1 \leq i_2 \leq 3n \ \wedge \ i_1 = i_2 \end{aligned}$$

```
      Algorithm incrAnalysisInsStmt
      Input : Statement S : The new statement inserted
      Output : Dependence Relations newDeps : The set of new dependences
1     newDep = φ
2     For all references S.k in S do
3       newDep = newDep ∪ FindDep(S.k, [S.k, s], S, U)
4     Endfor
5     For each dependence DepRels,S 1 in DepRelsS.1 such that Si ≠ S do
6       invalidDomain = Domain(DepRels,S 1)
7       For all old dependences as DepRelsS,Sj x such that Sj ≠ S
8         unCovered = Range(DepRelsS,Sj x \ invalidDomain)
9         DepRelsS,Sj x = DepRelsS,Sj x \ ¬invalidDomain
10        if ( x == 1 )
11          source = S
12        else
13          source = S ∪ Si
14        newDep = newDep ∪ FindDep( Sj.x, unCovered, source)
15      Endfor
16    Endfor
```

Figure 5: Incremental analysis - Insertion of assignment statement

3.0.4 Decrease in values read

The decrease of generated values invalidates certain dependences. This is expressed in the Lemma 3.9, the proof for which is obvious and omitted.

Lemma 3.9 *Let there be a dependence from* $S_i.1[i_1, s]$ *to* $S_j.x[i_2, s]$. *If on decrease of generated values, either* $S_i.1[i_1, s]$ *or* $S_j.x[i_2, s]$ *is not generated, then the dependence is not valid.* ∎

3.1 Insertion of Assignment Statement

The insertion of an assignment statement, say S increases the values generated in the program because of the write $S.1$ and the values read because of the references $S.k$ for $k > 1$. The algorithm for incremental analysis, **incrAnalysisInsStmt** is given in figure 5.

In the algorithm, the source function for all the references in S is calculated in statements 2-4. The references outside S, which have their dependence affected, and their new dependences are computed in statements 5-16. In each iteration of

the loop in statement 5, the dependences affected because of a new dependence as $DepRel_{S_i,S.1}$, and the resulting new dependences are computed. The variable `invalidDomain` contains the values of reference $S_i.1$ in the dependence $DepRel_{S_i,S.1}$. The source for a value $S_j.x[r,s]$ may be changed if its source say, $S_i.1[w,s]$ is in the set `invalidDomain`. All such $S_j.x[r,s]$ and their new source are calculated in statements 7-15. In each iteration of the loop in statement 7, the dependences for a reference $S_j.x$ having values in `invalidDomain` as source are updated for the new dependence. The variable `unCovered` computed in statement 8 contains the values of $S_j.x$ for which source is in `invalidDomain`. In statement 9, the dependence $DepRel_{S_i,S_j.x}$ is restricted to contain only those values of $S_i.1$ that are not in `invalidDomain`. This removes the part of dependence that may no longer be valid. Then the source for values in `notCovered` is calculated in statements 10-14. If $x = 1$ in $S_j.x$, then the source for the values of $S_j.1$ is only S, as given by the Theorem 3.6. If $x > 1$, then the source may be S or S_i as given by the Theorem 3.7. The source function for the value in `notCovered` is calculated in statement 14 by function FindDep. The function finds the source for the values of $S_j.x$ in set `unCovered`. The possible source references for these values is in the set **source**.

Example : Consider the program in figure 6(a), the dependences in the program are,

$$
\begin{aligned}
E \rightarrow S_1.1[i_2,j_2] \;&|\; 1 \le i_2, j_2 \le 20 \\
S_1[i_1,j_1] \rightarrow S_3.1[i_2,j_2] \;&|\; 1 \le i_2, j_2 \le 20 \;\wedge\; i_1 = i_2 \;\wedge\; j_1 = j_2 \\
S_1[i_1,j_1] \rightarrow S_3.2[i_2,j_2] \;&|\; 1 \le i_2 \le 20 \;\wedge\; 1 \le j_2 \le 10 \;\wedge\; i_1 = i_2 \;\wedge\; j_1 = 2j_2 \\
E \rightarrow S_3.2[i_2,j_2] \;&|\; 1 \le i_2 \le 20 \;\wedge\; 11 \le j_2 \le 20
\end{aligned}
$$

$$
S_3[i_1,j_1] \rightarrow S_4.1[i_2,j_2] \;|\; 1 \le i_2, j_2 \le 20 \;\wedge\; i_1 = i_2 \;\wedge\; j_1 = j_2 \qquad (2)
$$

If the statement S_2 is added at the position indicated, the resulting program is as shown in figure 6(b). The process of updating the dependences by the algorithm is given below.

First the source functions for $S_2.1$ and $S_2.2$ are calculated. The dependences for the references are,

$$
S_1[i_1,j_1] \rightarrow S_2.1[i_2,j_2] \;|\; 1 \le i_2 \le 20 \wedge\; 1 \le j_2 \le 10 \;\wedge\; i_1 = i_2 \;\wedge\; j_1 = 2j_2
$$

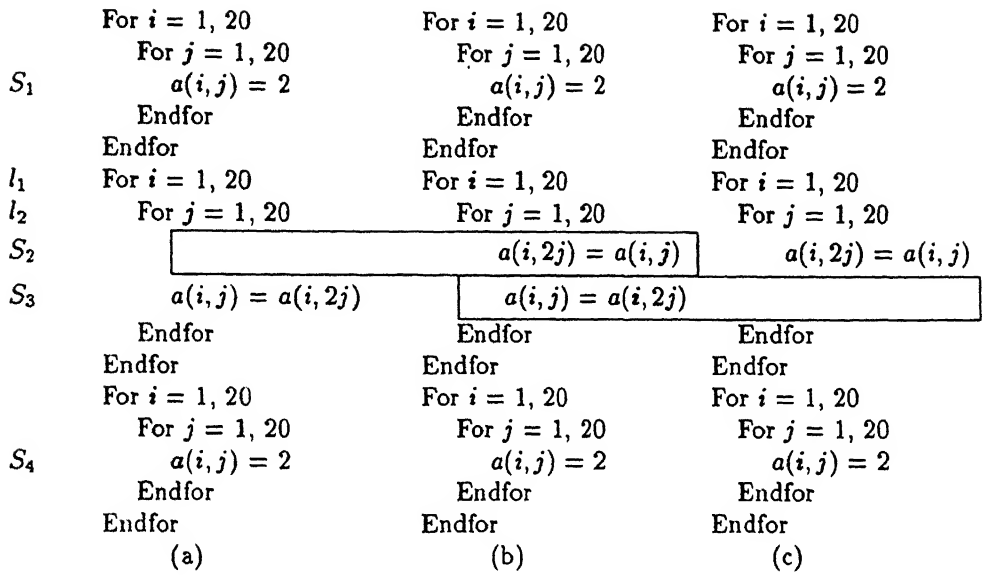


Figure 6: Example for addition/deletion of Assignment Statement

$$
\begin{aligned}
E \rightarrow S_2.1[i_2, j_2] \quad &| \quad 1 \leq i_2 \leq 20 \ \wedge \ 11 \leq j_2 \leq 20 \\
S_2[i_1, j_1] \rightarrow S_2.2[i_2, j_2] \quad &| \quad 1 \leq i_2, j_2 \leq 20 \ \wedge \ even(j_2) \ \wedge \ i_1 = i_2 \ \wedge \ 2j_1 = j_2 \\
S_1[i_1, j_1] \rightarrow S_2.2[i_2, j_2] \quad &| \quad 1 \leq i_2, j_2 \leq 20 \ \wedge \ odd(j_2) \ \wedge \ i_1 = i_2 \ \wedge \ j_1 = j_2
\end{aligned}
\tag{3}
$$

Consider the new dependence $DepRel_{S_1 S_2.1}$. The `invalidDomain` corresponding to this dependence is $S_1[i,j] \mid 1 \leq i,j \leq 20 \wedge even(j)$. The old dependence that may be possibly affected because of the new dependence is $DepRel_{S_1 S_3.1}$. The variable `unCovered` calculated in statement 6 is $S_3.1[i,j] \mid 1 \leq i,j \leq 20 \ \wedge \ even(j)$. The part the dependence that is not affected (calculated in statement 7) is,

$$
S_1[i_1, j_1] \rightarrow S_3.1[i_2, j_2] \quad | \quad 1 \leq i_2, j_2 \leq 20 \ \wedge \ odd(j_2) \ \wedge \ j_1 = j_2 \ \wedge \ i_1 = i_2
$$

Reference $S_3.1$ is a write hence, the possible source is only S. The new dependence for the values of $S_3.1$ given above is,

$$
S_2[i_1, j_1] \rightarrow S_3.1[i_2, j_2] \quad | \quad 1 \leq i_2, j_2 \leq 20 \ \wedge \ even(j_2) \ \wedge \ i_1 = i_2 \ \wedge \ 2j_1 = j_2
$$

```
      Algorithm incrAnalysisDelStmt
      Input : Statement S : The statement deleted
      Output : newDeps : The new dependences generated on deletion of S
1     newDep = φ
2     For each old dependence relation DepRel_{S,S.1} do
3        invalidDomain = Range(DepRel_{S,S.1})
4        For all dependences as DepRel_{SS.1}
5           invalidDomain = invalidDomain ∪ (Range(DepRel_{SS.1}) ∩ ¬Domain(DepRel_{SS.1}))
6        Endfor
7        For all old dependences as DepRel_{SS_j.x}
8           notCovered = Range(DepRel_{SS_j.x} \ invalidDomain)
9           newDep = newDep ∪ FindDep(S_j.x, notCovered, S_i)
10       Endfor
11    Endfor
12    For all dependence relation dep as DepRel_{S,S.x} or DepRel_{SS,y}
13       remove dep
14    Endfor
```

Figure 7: Incremental Analysis - Deletion of an assignment statement

Similarly for the reference $S_3.2$, the new dependences are

$$S_2[i_1, j_1] \rightarrow S_3.2[i_2, j_2] \quad | \quad 1 \leq i_2, j_2 \leq 20 \ \wedge \ i_1 = i_2 \ \wedge \ j_1 = j_2$$

3.2 Deletion of Assignment Statement

Deletion of an assignment statement, say S, decreases the values generated for all values generated by $S.1$, and the values read for all the values read by the references $S.x$, for $x > 1$. The incremental analysis on deletion of an assignment statement is done by the algorithm **incrAnalysisDelStmt**, given in figure 7.

Let the statement deleted be S. The algorithm computes new sinks for all the values having $S.1$ as the sink. In each iteration of the loop in statement 1, the new sinks for one source of $S.1$, say $S_i.1$, are computed. The variable `invalidDomain` contains values of $S.1$ having $S_i.1$ as the source. If there is a dependence from a value in `invalidDomain` to some value of $S_j.x$, then the source for the value of $S_j.x$ after deletion of S is $S_i.1$ (by Theorem 3.8). The variable `invalidDomain` is calculated in statements 3-6.

Let the source for a value of $S.1$, say $S.1[k_1, s]$, be $S_i.1[k, s]$ and sink be $S.1[k_2, s]$,

a values of $S.1$ itself. The new sink for $S_i.1[k,s]$ is $S.1[k_2,s]$. But $S.1[k_2,s]$ is also not generated after the deletion of S. So, the sink of $S.1[k_2,s]$ is the sink for the value of $S_i.1$. Again, the sink for $S.1[k_2,s]$ may be some value generated by $S.1$. This continues till the last write to the location referred by $S.1[k_1,s]$ in the dependence $DepRel_{SS.1}$. The last writes to the locations in dependence as $DepRel_{SS.1}$ are values that do not have any sinks in the dependence. These are the values in Range($DepRel_{SS.1}$) $\cap$ $\neg$Domain($DepRel_{SS.1}$). The last writes in the dependence is computed in the statements 4-6.

All the references $S_j.x$, that have $S.1$ as source are found in statement 7. The variable `notCovered` computed in statement 8 contains the values of $S_j.x$ for which the source is in `invalidDomain`. The sources for the values in `notCovered` is computed in statement 9. In statements 12-14 the invalid dependences as given by Lemma 3.9 are removed.

Example : Consider the program given in figure 6(b) again. The dependences in the program are,

$$
\begin{aligned}
E &\rightarrow S_1.1[i_2,j_2] \mid 1 \le i_2,j_2 \le 20 \\
S_1[i_1,j_1] &\rightarrow S_2.1[i_2,j_2] \mid 1 \le i_2 \le 20 \wedge 1 \le j_2 \le 10 \wedge i_1 = i_2 \wedge j_1 = 2j_2 \\
E &\rightarrow S_2.1[i_2,j_2] \mid 1 \le i_2 \le 20 \wedge 11 \le j_2 \le 20 \\
S_2[i_1,j_1] &\rightarrow S_2.2[i_2,j_2] \mid 1 \le i_2,j_1 \le 20 \wedge even(j_2) \wedge i_1 = i_2 \wedge 2j_1 = j_2 \\
S_1[i_1,j_1] &\rightarrow S_2.2[i_2,j_2] \mid 1 \le i_2,j_2 \le 20 \wedge odd(j_2) \wedge i_1 = i_2 \wedge j_1 = j_2 \\
S_2[i_1,j_1] &\rightarrow S_3.1[i_2,j_2] \mid 1 \le i_2,j_2 \le 20 \wedge even(j_2) \wedge i_1 = i_2 \wedge 2j_1 = j_2 \\
S_1[i_1,j_1] &\rightarrow S_3.1[i_2,j_2] \mid 1 \le i_2,j_2 \le 20 \wedge odd(j_2) \wedge i_1 = i_2 \wedge j_1 = j_2 \\
S_2[i_1,j_1] &\rightarrow S_3.2[i_2,j_2] \mid 1 \le i_2,j_2 \le 20 \wedge i_1 = i_2 \wedge j_1 = j_2 \\
S_3[i_1,j_1] &\rightarrow S_4.1[i_2,j_2] \mid 1 \le i_2,j_2 \le 20 \wedge i_1 = i_2 \wedge j_1 = j_2
\end{aligned}
\tag{4}
$$

If statement S_3 gets deleted, the resulting program is as shown in the figure 6(c). The process of updating the dependences from the algorithm **incrAnalysisDelStmt** is given below.

The dependences found in the statement 1 are, $DepRel_{S_1S_3.1}$ and $DepRel_{S_2S_3.1}$.

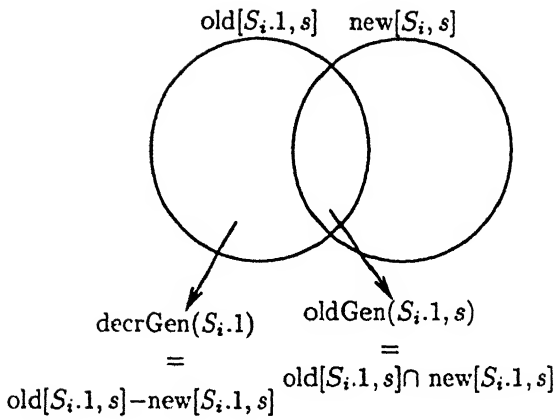


Figure 8: Values generated before and after insertion of loop

Consider the dependence $\mathrm{DepRel}_{S_1 S_3.1}$. The values in the variable invalidDomain are $S_1{:}1[i;j] \mid 1 \leq i;j \leq 20 \wedge \mathrm{odd}(j)$. The reference outside S_3 having $S_3{:}1$ as source, found in statement 7 is $S_4{:}1$. The values of $S_4{:}1$ for which new dependence is to be found is $\mathrm{Range}(\mathrm{DepRel}_{S_3 S_4.1} \setminus \mathrm{invalidDomain}) = S_4{:}1[i;j] \mid 1 \leq i;j \leq 20 \wedge \mathrm{odd}(j)$. The possible source for these values, according to the Theorem 3.8 is S_1. The new dependence $\mathrm{DepRel}_{S_1 S_4.1}$ is calculated in the statement number 9. Similarly, the new dependence formed for the case of $\mathrm{DepRel}_{S_2 S_3.1}$ in statement 1 is $\mathrm{DepRel}_{S_2 S_4.1}$.

The resulting dependences in the program are,

$$
\begin{aligned}
E &\rightarrow S_1{:}1[i_2;j_2] && \mid\ 1 \leq i_2;j_2 \leq 20 \\
S_1[i_1;j_1] &\rightarrow S_2{:}1[i_2;j_2] && \mid\ 1 \leq i_2 \leq 20 \wedge 1 \leq j_2 \leq 10 \wedge i_1 = i_2 \wedge j_1 = 2j_2 \\
E &\rightarrow S_2{:}1[i_2;j_2] && \mid\ 1 \leq i_2 \leq 20 \wedge 11 \leq j_2 \leq 20 \\
S_2[i_1;j_1] &\rightarrow S_2{:}2[i_2;j_2] && \mid\ 1 \leq i_2;j_1 \leq 20 \wedge \mathrm{even}(j_2) \wedge i_1 = i_2 \wedge 2j_1 = j_2 \\
S_1[i_1;j_1] &\rightarrow S_2{:}2[i_2;j_2] && \mid\ 1 \leq i_2;j_2 \leq 20 \wedge \mathrm{odd}(j_2) \wedge i_1 = i_2 \wedge j_1 = j_2 \\
S_2[i_1;j_1] &\rightarrow S_4{:}1[i_2;j_2] && \mid\ 1 \leq i_2;j_2 \leq 20 \wedge \mathrm{even}(j_2) \wedge i_1 = i_2 \wedge 2j_1 = j_2 \\
S_1[i_1;j_1] &\rightarrow S_4{:}1[i_2;j_2] && \mid\ 1 \leq i_2;j_2 \leq 20 \wedge \mathrm{odd}(j_1) \wedge i_1 = i_2 \wedge j_1 = j_2
\end{aligned}
$$

3.3 Insertion of a Loop

The insertion n of a loop l, with index j redefines all the references to variable j

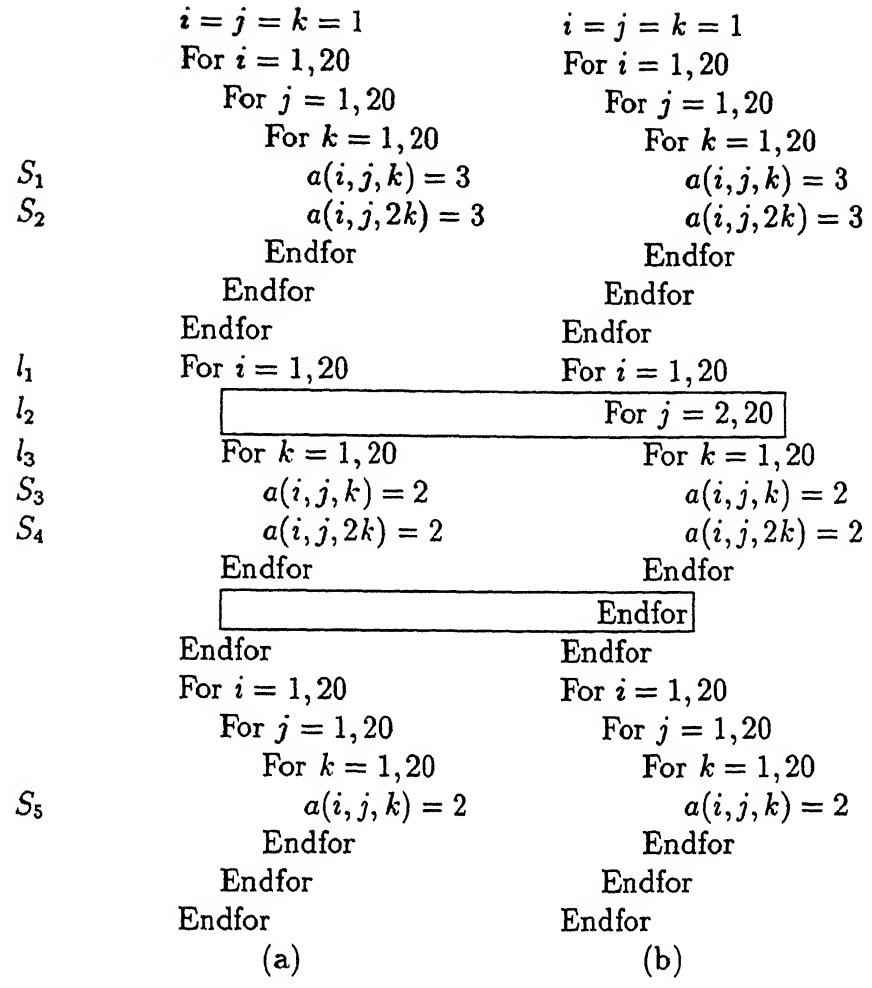


Figure 9: Example for addition of a loop

in the scope of j as the references to index j. It results in generation of certain new values for j. At the same time a few old values of j may not be regenerated. Let the new and the old values generated by the write in a statement S_i inside l be as shown in figure 8. Then the set of old values no longer generated after the insertion of the loop is obtained by complementing old by new. The set of old values which get re-generated is old $\cap$ new. The set of newly generated values is new $-$ old.

Example : Consider the program fragment in figure 9(a), the possible value of variable j before insertion of l_2 is only 1. After insertion of l_2, for the modified

program fragment in figure 9(b), the possible values of j are $2 \leq j \leq 20$. The new values generated by references $S_3.1$ and $S_4.1$ are, $a(i,j,k) \mid 1 \leq i,k \leq 20 \ \wedge \ 2 \leq j \leq 20$ and $a(i,j,2k) \mid 1 \leq i,k \leq 20 \ \wedge \ 2 \leq j \leq 20$ respectively. As for the decrease in the values generated, it may be noticed that, index j does not take value 1. Therefore result, the decrease in values generated by references $S_3.1$ and $S_4.1$ are, $a(i,j,k) \mid 1 \leq i,k \leq 20 \wedge j = 1$ and $a(i,j,2k) \mid 1 \leq i,k \leq 20 \wedge j = 1$ respectively.

The procedure for updating the dependences after insertion of a loop say l is,

- Calculate sources for the references inside the loop l.
- For references outside the loop l :

  update the dependences for decrease in the values generated.

  update the dependences for increase in the values generated.

Algorithm **incrAnalysisInsLoop** for incremental updating of dependences due to insertion of a loop is given in figure 10. The source functions for all the references inside the loop is computed in statement 8. The variable `decrValues`, is an array of sets. Each set, say `decrGen`($S_i.1$) contains values of $S_i.1$, that are not generated by $S_i.1$ after the loop is inserted. The variable `oldValues`, is also an array of sets. Each set say, `oldGen`($S_i.1$) contains the values of $S_i.1$, that are generated both before and after the insertion of loop. Pictorially, these variables are shown in the figure 8 for a reference $S_i.1$. A dependence dependence $DepRel_{S_iS_j.a}$ is affected due to decrease in generated values if,

1. S_j is outside the scope of l and
2. S_i is inside the scope of l and
3. Domain($DepRel_{S_iS_j.a}$) $\subseteq$ `decrValues`($S_i.1$)

The resulting new dependences are computed in the statement 10.

The kind of dependences that get affected as a result of increase in generated values after insertion of a loop are characterized in Lemma 3.11 and 3.10. The resulting new dependences are computed in the statement 13 using these characterization.

Lemma 3.10 *Let $S_i.1[i_1, s]$ write to a memory location M and a dependence $S_i.1[i_1, s] \rightarrow S_j.x[i_2, s]$ exist. Now suppose a loop l is inserted such that S_j is outside the scope of l, and $S_i.1[i_1, s]$ is generated after the insertion also, then all the writes to M in between $S_i.1[i_1, s]$ and $S_j.x[i_2, s]$ are new values generated.*

Proof : Let the dependence chain for the memory location M after the insertion of l be of form,

$$S_i.1[i_1, s] \rightarrow S_k.1[i_3, s], \ldots S_q.1[i_5, s], \ldots, S_n.1[i_4, s] \rightarrow S_j.x[i_2, s].$$

Let the value $S_q.1[i_5, s]$ also be generated before the insertion of l. Then it must be a write in between $S_i.1[i_1, s]$ and $S_j.x[i_2, s]$. But the existence of dependence $S_i.1[i_1, s] \rightarrow S_j.x[i_2, s]$ before the insertion of l implies that there is no write in between $S_i.1[i_1, s]$ and $S_j.x[i_2, s]$. This contradicts the above assumption. Hence, all the writes to M in between $S_i.1[i_1, s]$ and $S_j.x[i_2, s]$ are new values.∎

Lemma 3.11 *Let there be a dependence as $S_i.1[i_1, s] \rightarrow S_j.x[i_1, s]$. Suppose a loop l be inserted such that S_j is outside the scope of l. The source for $S_j.x[i_2, s]$ is changed due to increase in the generated values only if a dependence as $S_i.1[i_1, s] \rightarrow S_k.1[i_3, s]$ is formed and S_k is a new value formed after the insertion of l.*

Proof : Follows directly from Lemma 3.10. ∎

The function `updateDecrGen` updates the dependences for decrease in generated values after insertion of a loop. It finds the new sources for the values of $S_q.a$, in the dependence $DepRel_{S_pS_q.a}$. The variable `invalidDomain` in statements 14, gives the values of $S_p.1$ that were source for the values of $S_q.a$ before the insertion of the loop, but are not generated anymore. The dependence $DepRel_{S_pS_q.a}$ remains valid for the values of $S_p.1$ in the set `oldValues`($S_p.1$) after the decrease of values. These are the values of $S_p.1$ that get generated both before and after the decrease in the values. The variables are denoted pictorially in figure 3.3. The inner circle represents the values in Domain($DepRel_{S_iS_j.a}$). Lemma 3.12 gives the process to update the dependences for decrease in the generated values after the insertion of loop. The proof for correctness of Lemma 3.12 is obvious, however, we will give proof for termination of the process. Lemma 3.3 and 3.2 provide the basis of proof for Lemma 3.12.

Algorithm incrAnalysisInsLoop
Input : Loop : l /*The loop inserted*/
Output : Set of Dependence relations : newDep /*The set of new dependence relations*/
Modifies : The dependences relations of references affected.

1 For all assignment statements S_i do
2 Array of sets : `decrValues`$(S_i.1) = \phi$
3 Array of sets : `oldValues`$(S_i.1) =$ old$[S_i.1,s]$
4 Endfor
5 For all writes $S_i.x$ inside l do
6 If $(x == 1)$
7 `decrValues`$(S_i.x) =$ old $[S_i.x, s]$ − new$[S_i.x, s]$
8 `oldValues`$(S_i.x) =$ new$[S_i.x, s] \cap$ old$[S_i.x, s]$
9 Endif
10 newDep = newDep ∪ FindDep$(S.x, [S.x, s], S, U)$
11 Endfor
12 ∀old $DepRel_{S_i S_j.a}$ such that S_i is inside l and S_j outside l
updateDecrGen$(DepRel_{S_i S_j.a})$
13 ∀$DepRel_{S_i S_j.a}$ such that S_j is outside l and
∃ new$DepRel_{S_i S_k.1}$ such that S_k inside l
updateIncrGen$(DepRel_{S_i S_k.a} \setminus$ `oldValues`$(S_i))$

updateDecrGen$(DepRel_{S_p S_q.a})$
14 Set of values : `invalidDomain` = Domain$(DepRel_{S_p S_q.a}) \cap$ decrValues$(S_p.1)$
15 For all old dependence as $DepRel_{S_k S_p.1}$
16 $DepRel_{S_k S_q.a}$ = decrGen$(DepRel_{S_p S_q.a}, DepRel_{S_k S_p.1},$ `invalidDomain`)
17 If ($DepRelS_k S_q.a \neq \phi$)
18 updateDecrGen$(DepRel_{S_k S_q.a})$
19 Endif
20 Endfor
21 $DepRel_{S_p S_q.a} = DepRel_{S_p S_q.a} \setminus$ `oldValues`$(S_p.1)$

updateIncrGen$(DepRel_{S_p S_q.a})$
22 For all new dependences $DepRel_{S_p S_k.1}$ such that $S_k.1$ inside l
23 $DepRel_{S_k S_q.a}$ = incrGen$(DepRel_{S_p S_q.a}, DepRel_{S_p S_k.1},)$
24 If ($DepRel_{S_k S_q.a} \neq \phi$)
25 updateIncrGen$(DepRel_{S_k S_q.a})$
26 Endif
27 $DepRel_{S_p S_q.a} = DepRel_{S_p S_q.a} / \neg$ Range$(DepRel_{S_k S_q.a})$
28 Endfor
29 `newDep` = `newDep` ∪ $DepRel_{S_p S_q.a}$

Figure 10: Incremental Analysis - Insertion of Do loop

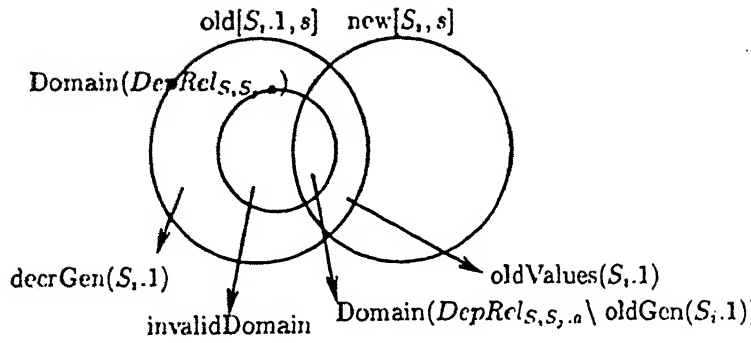


Figure 10: Domain of an old dependence

Lemma 3.12 *Let there be a dependence as* $S_i.1[i_1, s] \rightarrow S_j.1[i_2, s]$. *After deletion of a loop* l *such that* S_j *is outside* l, *if* $S_i.1[i_1, s]$ *is not generated, the new source for* $S_j.x[i_2, s]$ *can be computed by a finite number of recursive updates for decrease in the generated values.*

Proof : Let $S_i.1[i_1, s]$ write to memory location M. The reference $S_j.a[i_2, s]$ writes to M if $a = 1$, or reads from M if $a \neq 1$.

From Lemma 3.2, Entry E is the first write to M. The dependence chain corresponding to memory location M is of the form,

$$E \rightarrow S_p.1[i_4, s], S_p.1[i_4, s] \rightarrow S_q.1[i_5, s].....S_r[i_6, s] \rightarrow S_i[i_1, s], S_i[i_1, s] \rightarrow S_j.a[i_2, s].$$

If $S_i.1[i_1, s]$ is not generated after insertion of loop l. Theorem 3.8 should be applied for decrease in the generated values. This implies, the last two dependences in the dependence chain must be merged. More precisely, the dependence chain corresponding to memory location M are after the first updating is,

$$E \rightarrow S_p.1[i_4, s], S_p.1[i_4, s] \rightarrow S_q.1[i_5, s].....S_r[i_6, s] \rightarrow S_j.a[i_2, s].$$

According to the theorem, the process of updating stops on detecting a value that is generated after the decrease in the generated values. When the source say $S_i.1$ is outside l, then there is no change in the values generated by $S_i.1$. Hence, the updating process stops after single application of Theorem 3.8. If the source is

inside l and is re-generated after the insertion the process terminates. However, when there are no such source references in between, the repeated application of Theorem 3.8 on the dependence chain for M will eventually lead to,

$$E \rightarrow S_j.a[i_2, s].$$

at one stage. From Lemma 3.3, there is no change in the values generated by E. Hence, for a finite length program the process will eventually terminate.∎

The function *decrGen* updates the dependences according to Theorem 3.8, and the recursive call to updateDecrGen in statement 18 updates the new dependences generated by *decrGen* for decrease in the generated values. The part of the dependence, that is not valid after the decrease in values generated is removed from $DepRel_{S_pS_q.a}$ in statement 21.

The incremental updates for increase in the values generated is done by the function updateIncrGen. This is also a recursive function as *updateDecrGen*. It finds the new sources for the values of $S_q.a$ in the dependence $DepRel_{S_pS_q.a}$.

The dependences in newDeps are new dependences generated after the insertion of l. A new dependence $DepRel_{S_pS_k.1}$, implies that some values of $S_p.1$ are overwritten by the reference $S_k.1$. Hence, the dependence $DepRel_{S_pS_q.a}$ must be updated for increase in the values generated by $S_k.1$. A dependence $DepRel_{S_pS_q.a}$ is updated for each of the new dependence as $DepRel_{S_pS_k.1}$ in newDeps in statements 22-28. The updating is done by the function *incrGen* in statement 23 as given by Theorem 3.6 and 3.7. The function forms a new dependence $DepRel_{S_kS_q.a}$. This dependence contains the values of $S_q.a$ for which the new source is S_k. The values of $S_j.a$, for which the source is changed are removed from $DepRel_{S_pS_q.a}$ in statement 27.

The values of $S_k.1$ may further be overwritten before referred to by $S_q.a$ as given in the proof for the Lemma 3.11. Hence, the new dependence must also updated with function updateIncrGen as in statement 25. The dependence $DepRel_{S_pS_q.a}$ remaining in the statement 29 is valid after the insertion of the loop l, and is added to newDeps.

Example : Consider the program given in figure 9(a). The dependences in the program are,

$$S_2[i_1, j_1, k_1] \rightarrow S_1.1[i_2, j_2, k_2] \mid 1 \leq i_2, j_2, k_2 \leq 20 \wedge even(k_2) \wedge i_1 = i_2 \wedge$$

$$
\begin{aligned}
& & & j_1 = j_2 \ \wedge \ 2k_1 = k_2 \\
E \rightarrow S_1.1[i_2, j_2, k_2] & \mid & & 1 \leq i_2, j_2, k_2 \leq 20 \ \wedge \ odd(k_2) \\
E \rightarrow S_2.1[i_2, j_2, k_2] & \mid & & 1 \leq i_2, j_2, k_2 \leq 20 \\
S_4[i_1, k_1] \rightarrow S_3.1[i_2, k_2] & \mid & & 1 \leq i_2, k_2 \leq 20 \ \wedge \ even(k_2) \ \wedge \\
& & & i_1 = i_2 \ \wedge \ 2k_1 = k_2 \\
S_1[i_1, j_1, k_1] \rightarrow S_3.1[i_2, k_2] & \mid & & 1 \leq i_2, k_2 \leq 20 \ \wedge \ odd(k_2) \ \wedge \ j_1 = 1 \ \wedge \\
& & & i_1 = i_2 \ \wedge \ j_1 = j_1 \ \wedge \ k_1 = k_2 \\
S_1[i_1, j_1, k_1] \rightarrow S_4.1[i_2, k_2] & \mid & & 1 \leq i_2 \leq 20 \ \wedge \ j_1 = 1 \ \wedge \ 1 \leq k_2 \leq 10 \ \wedge \\
& & & i_1 = i_2 \ \wedge \ j_1 = j_1 \ \wedge \ k_1 = 2k_2 \\
S_2[i_1, j_1, k_1] \rightarrow S_4.1[i_2, k_2] & \mid & & 1 \leq i_2 \leq 20 \ \wedge \ j_1 = 1 \ \wedge \ 11 \leq k_2 \leq 20 \wedge \\
& & & i_1 = i_2 \ \wedge \ j_1 = j_1 \ \wedge \ k_1 = k_2 \\
S_3[i_1, k_1] \rightarrow S_5.1[i_2, j_2.k_2] & \mid & & 1 \leq i_2, k_2 \leq 20 \ \wedge \ j_2 = 1 \ \wedge \ i_1 = i_2 \ \wedge \ k_1 = k_2 \\
S_1[i_1, j_1, k_1] \rightarrow S_5.1[i_2, j_2, k_2] & \mid & & 1 \leq i_2, k_2 \leq \ 20 \ \wedge \ 2 \leq j_2 \leq \ 20 \ \wedge \ i_1 = i_2 \\
& & & \wedge \ j_1 = j_2 \ \wedge \ k_1 = k_2
\end{aligned}
$$

The process of updating dependences by the algorithm incrAnalysisInsLoop on insertion of the loop l_2 is given below.

The `decrValues` and `oldValues` for the statements S_3 and S_4 are,

$$
\begin{aligned}
\texttt{decrValues}(S_3.1) = S_3[i, j, k] & \mid & & 1 \leq i, k \leq 20 \ \wedge \ j = 1 \\
\texttt{decrValues}(S_4.1) = S_4[i, j, 2k] & \mid & & 1 \leq i, k \leq 20 \ \wedge \ j = 1 \\
\texttt{oldValues}(S_3.1) & = & & \phi \\
\texttt{oldValues}(S_4.1) & = & & \phi
\end{aligned}
$$

The dependences for the references inside the loop inserted are,

$$
\begin{aligned}
S_1[i_1, j_1, k_1] \rightarrow S_3.1[i_2, j_2, k_2] & \mid & & 1 \leq i_2, k_2 \leq 20 \ \wedge \ odd(k_2) \ \wedge \ 2 \leq j_2 \leq 20 \ \wedge \\
& & & i_1 = i_2 \ \wedge \ j_1 = j_2 \ \wedge \ k_1 = k_2 \\
S_4[i_1, j_1, k_1] \rightarrow S_3.1[i_2, j_2, k_2] & \mid & & 1 \leq i_2, k_2 \leq 20 \ \wedge \ even(k_2) \ \wedge \ 2 \leq j_2 \leq 20 \ \wedge \\
& & & i_1 = i_2 \ \wedge \ j_1 = j_2 \ \wedge \ 2k_1 = k_2
\end{aligned}
$$

$$S_1[i_1, j_1, k_1] \rightarrow S_4.1[i_2, j_2, k_2] \mid 1 \leq i_2 \leq 20 \wedge 2 \leq j_2 \leq 20 \wedge 1 \leq k_2 \leq 10 \wedge i_1 = i_2 \wedge j_1 = j_2 \wedge k_1 = 2k_2$$

$$S_2[i_1, j_1, k_1] \rightarrow S_4.1[i_2, j_2, k_2] \mid 1 \leq i_2 \leq 20 \wedge 2 \leq j_2 \leq 20 \wedge 11 \leq k_2 \leq 20 \wedge i_1 = i_2 \wedge j_1 = j_2 \wedge k_1 = k_2$$

The dependences are then updated for the decrease in the values generated. $DepRelS_3S_5.1$ is the only dependence affected as a result of decrease in the values. The function updateDecrGen is called with the dependence $DepRel_{S_3S_5.1}$. The `invalidDomain` is the complete values of S_3 in the dependence, that is,

$$\texttt{invalidDomain} = S_3.1[i, j, k] \mid 1 \leq i, k \leq 20 \wedge j = 1.$$

The variable dep $= \phi$ because, oldValues$(S_3.1) = \phi$. The old sources for $S_3.1$ corresponding to the values in unCovGen are $S_1.1$ and $S_4.1$. Dependences $DepRel_{S_1S_5.1}$ and $DepRel_{S_4S_5.1}$ are formed in statement 22. The dependences formed above are are,

$$S_1[i_1, j_1, k_1] \rightarrow S_5.1[i_2, j_2, k_2] \mid 1 \leq i_2, k_2 \leq 20 \wedge odd(k_2) \wedge j_2 = 1 \wedge i_1 = i_2 \wedge j_1 = j_2 \wedge k_1 = k_2$$

$$S_4[i_1, j_1, k_1] \rightarrow S_5.1[i_2, j_2, k_2] \mid 1 \leq i_2, k_2 \leq 20 \wedge even(k_2) \wedge j_2 = 1 \wedge i_1 = i_2 \wedge j_1 = j_2 \wedge 2k_1 = k_2$$

The new dependences above are updated by the function updateDecrGen in statement 23. Consider the case of $DepRel_{S_1S_5.1}$. There is no change in the values generated by $S_1.1$. Hence, there are no new dependences formed in statement 22, and the dependence is valid after the decrease in the generated values. Consider the case of $DepRel_{S_4S_5.1}$. The values in Domain$(DepRel_{S_4S_5.1})$ are no longer generated. The variable `invalidDomain` contains all the values of $S_4.1$ in the dependence $DepRep_{S_4S_5.1}$. The source reference for these values of $S_4.1$ is $S_1.1$. Hence, the source for the values of $S_5.1$ in the dependence above is $S_5.1$. The dependence formed as a result of this is,

$$S_1[i_1, j_1, k_1] \rightarrow S_5.1[i_2, j_2, k_2] \mid 1 \leq i_2, k_2 \leq 20 \wedge even(k_2) \wedge j_2 = 1 \wedge i_1 = i_1 \wedge j_1 = j_2 \wedge k_1 = k_2$$

The function updateDecrGen is called with the dependence above. There is no change in the values generated by $S_1.1$, hence there are no dependences formed in 22, and the complete dependence $DepRel_{S_1S_5.1}$ is valid. The dependence $DepRel_{S_1S_5.1}$ after the updating for decrease in the generated values is,

$$S_1[i_1, j_1, k_1] \rightarrow S_5.1[i_2, j_2, k_2] \quad | \quad 1 \leq i_2, j_2, k_2 \leq 20 \ \wedge \ i_1 = i_2 \ \wedge \ j_1 = j_2 \ \wedge \ k_1 = k_2$$

The dependences affected as a result of increase in the values are updated in statement 16. The only dependence affected because of increase in the values generated is $S_5.1$. The function updateIncrGen is called with $DepRel_{S_1S_5.1}$. The statement S_1 is not affected by the insertion of the loop, hence the function updateIncrGen is called with complete dependence $DepRel_{S_1S_5.1}$. Some of the values of $S_1.1$ in the dependence $DepRel_{S_1S_5.1}$ are overwritten by the references $S_3.1$ and $S_4.1$ as given by the dependences $DepRel_{S_1S_3.1}$ and $DepRel_{S_1S_4.1}$ in newDeps. The dependences $DepRel_{S_3S_5.1}$ and $DepRel_{S_3S_5.1}$ formed in the statement 34 are,

$$\begin{aligned} S_3[i_1, j_1, k_1] \rightarrow S_5.1[i_2, j_2, k_2] \quad | \quad & 1 \leq i_2, k_2 \leq 20 \ \wedge \ odd(k_2) \ \wedge \ 2 \leq j_2 \leq 20 \ \wedge \\ & i_1 = i_2 \ \wedge \ j_1 = j_2 \ \wedge \ k_1 = k_2 \\ S_4[i_1, j_1, k_1] \rightarrow S_5.1[i_2, j_2, k_2] \quad | \quad & 1 \leq i_2, k_2 \leq 20 \ \wedge \ even(k_2) \ \wedge \ 2 \leq j_2 \leq 20 \ \wedge \\ & i_1 = i_2 \ \wedge \ j_1 = j_2 \ \wedge \ 2k_1 = k_2 \end{aligned}$$

The values in the dependences above are removed from the dependence $DepRel_{S_1S_5.1}$ in statement 29. The resulting dependence $DepRel_{S_1S_5.1}$ is,

$$S_1[i_1, j_1, k_1] \rightarrow S_5.1[i_2, j_2, k_2] \quad | \quad 1 \leq i_2, k_2 \leq 20 \ \wedge \ j_2 = 1 \ \wedge \ i_1 = i_2 \ \wedge \ j_1 = j_2 \ \wedge \ k_1 = k_2.$$

The new dependences $DepRel_{S_3S_5.1}$ and $DepRel_{S_4S_5.1}$ formed above are again updated for decrease in values by the function updateIncrGen in statement 35. For $DepRel_{S_3S_5.1}$, there is no further overwriting of the values. For the case of dependence $DepRel_{S_4S_5.1}$, the values of $S_4.1$ in $DepRel_{S_4S_5.1}$ are overwritten by $S_3.1$ as given by the dependence $DepRel_{S_4S_3.1}$. The dependence formed in statement 27 is,

$$S_3[i_1, j_1, k_1] \rightarrow S_5.1[i_2, j_2, k_2] \quad | \quad 1 \leq i_2, k_2 \leq 20 \ \wedge \ even(k_2) \ \wedge \ 2 \leq j_2 \leq 20 \ \wedge \ i_1 = i_2 \ \wedge \ j_1 = j_2 \ \wedge \ k_1 = k_2$$

This function updateIncrGen is again called with the dependence above. There is no more overwriting of values written by $S_3.1$ in the dependence. Hence there no more call to the function updateIncrGen. The dependences for the reference $S_5.1$ after updating for increase and decrease in the generated values is,

$$\begin{aligned} S_3[i_1, j_1, k_1] \rightarrow S_5.1[i_2, j_2, k_2] \quad &| \quad 1 \leq i_2 \leq 20 \ \wedge \ 2 \leq j_2 \leq 20 \ \wedge \ 1 \leq k_2 \leq 20 \ \wedge \\ &\quad i_1 = i_2 \ \wedge \ j_1 = j_2 \ \wedge \ k_1 = k_2 \\ S_1[i_1, j_1, k_1] \rightarrow S_5.1[i_2, j_2, k_2] \quad &| \quad 1 \leq i_2, k_2 \leq 20 \ \wedge \ j_2 = 1 \ \wedge \\ &\quad i_1 = i_2 \ \wedge \ j_1 = j_2 \ \wedge \ k_1 = k_2 \end{aligned}$$

3.4 Deletion of Loop

The deletion of a loop l with index j, redefines all the references to the index i in the scope of l, as the references to the variable j. It results in generation of certain new values for variable j. At the same time a few old values of index j may not be generated for variable j. As a result of this there is both increase and decrease in the generated values. The incremental analysis procedure on deletion of a loop is similar to the analysis for insertion of a loop.

Example : Consider the program in figure 9(b). The values generated by references $S_3.1$ and $S_4.1$ are,

$$\begin{aligned} a(i, j, 2k) \quad &| \quad 1 \leq i, k \leq 20 \ \wedge \ 2 \leq j \leq 20 \\ a(i, j, k) \quad &| \quad 1 \leq i, k \leq 20 \ \wedge \ 2 \leq j \leq 20. \end{aligned}$$

After deletion of the loop l_2, the possible values of the variable j is only 1 hence, the values given above are not generated. There is decrease in the values generated for all the values above. The new values generated by the references $S_3.1$ and $S_4.1$ are,

$$\begin{aligned} a(i, j, 2k) \quad &| \quad 1 \leq i, k \leq 20 \ \wedge \ j = 1 \\ a(i, j, k) \quad &| \quad 1 \leq i, k \leq 20 \ \wedge \ j = 1. \end{aligned}$$

These values were not generated before the deletion of the loop, hence there is an

increase in the values generated by the statements for these values. The process of updating the dependences on deletion of loop l_2 is given below.

The variables `decrValues` and `oldValues` for the statements inside l_2 are,

$$\begin{aligned}
\texttt{decrValues}(S_3.1) &= S_3[i,j,k] \quad | \quad 1 \leq i,k \leq 20 \ \wedge \ 2 \leq j \leq 20 \\
\texttt{decrValues}(S_4.1) &= S_4[i,j,2k] \quad | \quad 1 \leq i,k \leq 20 \ \wedge \ 2 \leq j \leq 20 \\
\texttt{oldValues}(S_3.1) &= \phi \\
\texttt{oldValues}(S_4.1) &= \phi
\end{aligned}$$

The new dependences for the references inside the loop l_2 are,

$$\begin{aligned}
S_1[i_1,j_1,k_1] \rightarrow S_3.1[i_2,k_2] \quad | \quad & 1 \leq i_2,k_2 \leq 20 \ \wedge \ odd(k_2) \ \wedge \ j_1 = 1 \ \wedge \\
& i_1 = i_2 \ \wedge \ k_1 = k_2 \\
S_4[i_1,k_1] \rightarrow S_3.1[i_2,k_2] \quad | \quad & 1 \leq i_2,k_2 \leq 20 \ \wedge \ even(k_2) \ \wedge \ i_1 = i_2 \ \wedge \ 2k_1 = k_2 \\
S_1[i_1,j_1,k_1] \rightarrow S_4.1[i_2,k_2] \quad | \quad & 1 \leq i_2 \leq 20 \ \wedge \ j_1 = 1 \ \wedge \ 1 \leq k_2 \leq 10 \ \wedge \\
& i_1 = i_2 \ \wedge \ k_1 = 2k_2 \\
S_2[i_1,j_1,k_1] \rightarrow S_4.1[i_2,k_2] \quad | \quad & 1 \leq i_2 \leq 20 \ \wedge \ j_1 = 1 \ \wedge \ 11 \leq k_2 \leq 20 \ \wedge \\
& i_1 = i_2 \ \wedge \ k_1 = k_2
\end{aligned}$$

First the dependences are updated for the decrease in the values generated. The only dependence affected is $DepRel_{S_1 S_5.1}$. Consider the call of the function updateDecrGen with the dependence $DepRel_{S_3 S_5.1}$. Similar to the case in insertion of the loop, the possible writes are S_1 and S_4. The dependences $DepRel_{S_1 S_5.1}$ and $DepRel_{S_4 S_5.1}$ are formed and are updated for further decrease in values. In case of dependence $DepRel_{S_1 S_5.1}$, there are no more changes and the dependence is valid. In the invocation of updateDecrGen with $DepRel_{S_4 S_5.1}$, the dependence $DepRel_{S_1 S_5.1}$ is formed and, again the function updateDecrGen is called with this. There are no more changes to the dependence. The resulting dependence after updating for decrease in the generated values is,

$$\begin{aligned}
S_1[i_1,j_1,k_1] \rightarrow S_5.1[i_2,j_2,k_2] \quad | \quad & 1 \leq i_2 \leq 20 \ \wedge \ 2 \leq j_2 \leq 20 \ \wedge 1 \leq k_2 \leq 20 \ \wedge \\
& i_1 = i_2 \ \wedge \ j_1 = j_2 \ \wedge \ k_1 = k_2
\end{aligned}$$

Then the dependences are updated for increase in the values generated. Again the process for updating is same as for insertion of loop l_2 given before. The resulting dependence after updating for increase in the generated values are,

$$S_3[i_1, k_1] \rightarrow S_5.1[i_2, j_2, k_2] \quad | \quad 1 \leq i_2, k_2 \leq 20 \ \wedge \ j_2 = 1 \ \wedge \ i_1 = i_2 \ \wedge \ k_1 = k_2$$

Chapter 4

Implementation

This chapter deals with the algorithms we have implemented. The incremental algorithms presented in this chapter are implemented over the UMCP version of the **Tiny tool**. The algorithm "Lazy Array Data-Flow Dependence Analysis" proposed by Vadim Maslov [7] is used for exhaustive analysis. The algorithm with a small modification to accept different inputs is given in the Appendix A.

The concept of "Dependence crossing a statement" is defined below. This concept is used in the incremental form of the exhaustive algorithm.

4.0.1 Dependence Crossing An Assignment Statement

A dependence $S_i.1[i_1, s] \rightarrow S_j.x[i_2, s]$ crosses an instance of an assignment statement $S[i_3, s]$ if, the instance $S[i_3, s]$ is executed in-between $S_i[i_1, s]$ and $S_j[i_2, s]$. That is if, $S_i[i_1, s] \ll S[i_3, s] \ll S_j[i_2, s]$.

A dependence relation $DepRel_{S_i S_j.x}$ crosses an assignment statement S if, there exists an instance of the dependence $S_i.1[i_1, s] \rightarrow S_j.x[i_2, s]$, and an instance of an assignment statement S, say $S[i_3, s]$ such that $S_i.1[i_1, s] \rightarrow S_j.x[i_2, s]$ crosses $S[i_3, s]$. An approximation to the definition above can be defined as given below.

A dependence $DepRel_{S_i S_j.x}$ crosses an assignment statement S if,

- it is loop independent and $S_i \ll S \ll S_j$ or
- it is loop carried and S is inside the loop carrying the dependence.

This definition is conservative. It may say that a dependence crosses a statement when it does not. However, if the dependence crosses an assignment statement, the definition always says so.

4.0.2 Dependence Crossing Loop

Consider that *For* and *Endfor* statements of every loop are executed before and after each iteration of the loop respectively. A dependence crosses the loop l if, the dependence crosses either *For* or the *Endfor* statement of the loop. An approximation to the above definition in terms of the loop carrying the dependence can be defined as below.

A dependence $DepRel_{S,S_j.x}$ crosses a loop l_n if,

- it is loop independent dependence carried by loop l, and l_n is inside l.
- it is loop carried dependence carried by loop l, and l_n is either l or inside l.

This definition is also a conservative one.

4.1 Insertion Of An Assignment Statement

The insertion of an assignment statement increases the values generated in a program as given in chapter 3. The algorithm **IncrAnalysisInsStmt** given in figure 12 updates the dependences on insertion of an assignment statement. The analysis on insertion of a statement S consists of the following steps,

1. Find the source functions for all the references in the statement S.
2. Find the references outside S say, $S_j.x$ such that the sources functions of $S_j.x$ are affected.
3. Find all the values of reference $S_j.x$ for which the sources are to be computed.
4. Compute the new sources for the values of $S_j.x$ found in (3).

The variables used in the algorithm are,

Algorithm : IncrAnalysisInsStmt
Input : Statement S : The statement inserted
Output : The updated dependences

1 `source` = ϕ
2 $\forall S.k$, FindDep($S.k, S, [S.k, s]$, U)
3 $\forall\ DepRel_{s,s} \epsilon DepRels_{S,1}$ `source` = `source` $\cup\ S_i$
4 For each $S_j.x$ such that $((S_j \neq S)$ && $(\exists DepRel_{S,S_j,x}$ and $S_i \epsilon$ `source`$))$
5 $\forall DepRel_{S_p,S_j,x}$ such that $(S_p \epsilon$ `source`$)$ && $(DepRel_{S_p,S_j,x}$ crosses S)
6 `notCovered` = `notCovered` $\cup$ Range$(DepRel_{S_p,S_j,x})$
7 If ($S_j \ll S$)
8 beginStmt = S
9 Else
10 beginStmt = lastCommonLoop(S, S_j)
11 FindDepI($S_j.x$, `notCovered`, `beginStmt`, `source`)
12 Endfor

Figure 12: Incremental algorithm for insertion of assignment statement

`source` : The set of all the source references of $S.1$.

`notCovered` : The set of values of the reference being updated.

`beginStmt` : The statement from which the source functions for the affected values is to be recomputed.

The source functions for the references in statement S are computed in statement 2, and the set `source` is computed in statement 3. The references outside S that have dependences affected are found in 4.

A reference outside S is considered to have its dependences affected if, one of its dependence is affected as a result of the increase in values generated. A dependence $DepRel_{s,s_j,x}$ is affected as a result of increase in the generated values if S_i is source for some of the values of $S.1$ (as in chapter 3), and the dependence crosses the statement S. If the dependence does not cross the statement S, there are no writes of $S.1$ in between values of $S_i.1$ and $S_j.x$. Hence, the dependence is not affected.

The values of $S_j.x$ that become uncovered are computed in statement 5. If a dependence is affected, all the values of $S_j.x$ in the dependence are considered to

become uncovered. This is a conservative approximation to the values that become uncovered (as given in chapter 3). A reference $S_j.x$ may have more than one such dependences affected, all such dependences affected are computed in statement 5. All the values of $S_j.x$ in these dependences are added to `notCovered` in statement 6.

The new sources for values in `notCovered` are computed in statements 7-11. The new sources are computed by the function *FindDep1*. The function *FindDep1* is similar to function *FindDep* given in Appendix A. However, the difference between the two functions are,

1. In function **FindCoverSingleWrite**, dependence from a source S_i to $S_j.x$ is computed only if the source is $S.1$. For other writes, the range of old dependence $DepRel_{S,S_j.x}$ is restricted to the values in `notCovered`.

   The dependence from the old sources need not be recomputed because, there is no decrease in the generated values by any statement. As a result, at any stage in the computing the source function, there are no new values of $S_j.x$ in the `notCovered` as compared to the exhaustive analysis. Hence, the dependences from old statements need not be recomputed. However, new values from $S.1$ may remove some values in `notCovered` as compared to the exhaustive analysis. Hence, the old dependences are to be restricted to the values in `notCovered`.

2. In the function **FindCoverMultipleWrite** only the dependences with new write as source are to be updated with the function $RelMax_2$. The old dependences are already updated with the function $RelMax_2$, hence need not be updated again. The function is given in the figure 13.

The statement from which the search should start is found in statements 7-10. If S_j appears after S lexically, then there can be a loop independent dependence from $S.1$ to $S_j.x$ hence, the search starts from statement S. If the statement S_j appears before S lexically, then there can be no loop independent dependence from $S.1$ to $S_j.x$. The search should start from finding the loop carried dependences carried by the last common loop between S and S_j.

```
Function FindCoverMultipleWrite(reference S_j.x, set possibleSources, DNF notCovered,
                                Statement loop, Flag carriedFlag)
1  cMax = φ
2  Forall writes S_i.1 such that S_i ∈ possibleSources && S_i ≠ statement inserted
3     wrMax = wrMax ∪ (DepRel_{S_i,S_j.x}/notCovered)
4  cMax = FindCoverSingleWrite(S_j.x, S_1.1, notCovered, l_n, carriedFlag)
5  wrMax = RelMax_2(wrMax, cMax)
6  Endfor
7  return wrMax
```

Figure 13: Function FindCovereMultipleWrite for Incremental Analysis

Example : Consider the program given in figure 6(a). The dependences in the program are given in equation 2. Suppose a new statement, S_2 is added at the position indicated, the program is as shown in figure 6(b). The process of updating the dependences by the algorithm **IncrAnalysisInsStmt** is given below.

The dependences for references $S_2.1$ and $S_2.2$ are computed in statement 2. The dependences for these references are as in equation 3. The set source = { $S_1.1$, E}. The references outside S_2, that have their dependence possibly affected as found in statement 4 are $S_3.1$, $S_4.1$ and $S_3.2$.

Consider the case of reference $S_3.1$. The dependence affected is only $DepRel_{S_1 S_3.1}$. The notCovered computed for the reference in statement 5 is $S_3.1[i,j] \mid 1 \leq i,j \leq 20$. Statement S_3 appears after S_2 lexically hence, the beginStmt is S_2. The new sources for values in notCovered computed by the function *FindDep1* are,

$$S_2.1[i_1,j_1] \rightarrow S_3.1[i_2,j_2] \quad | \quad 1 \leq i_2,j_2 \leq 20 \ \wedge \ even(j_2) \ \wedge \ i_1 = i_2 \ \wedge \ 2j_1 = j_2$$

$$S_1.1[i_1,j_1] \rightarrow S_3.1[i_2,j_2] \quad | \quad 1 \leq i_2,j_2 \leq 20 \ \wedge \ odd(j_2) \ \wedge \ i_1 = i_2 \ \wedge \ j_1 = j_2$$

Similarly for the reference $S_4.1$, the dependence affected is $DepRel_{S_3 S_4.1}$, and notCovered is $S_4.1[i,j] \mid 1 \leq i,j \leq 20$. The statement S_4 appears after S_2 hence, the beginStmt is S_2. The new sources for the values in notCovered computed by *FindDep1* are,

$$S_3.1[i_1,j_1] \rightarrow S_4.1[i_2,j_2] \quad | \quad 1 \leq i_2,j_2 \leq 20 \ \wedge \ i_1 = i_2 \ \wedge \ j_1 = j_2$$

Similarly the new source functions for the reference $S_3.2$ is

$$S_2.1[i_1,j_1] \rightarrow S_3.2[i_2,j_2] \quad | \quad 1 \leq i_2,j_2 \leq 20 \ \wedge \ i_1 = i_2 \ \wedge \ j_1 = j_2.$$

4.2 Deletion of Assignment Statement

The deletion of an assignment statement say, S decreases the generated values in the program as given in chapter 3. The algorithm for incremental analysis on deletion of statement, **IncrAnalysisDelStmt** is given in figure 14. The process of updating dependences on deletion of assignment statement consists of the following steps,

1. Find all the references outside S say, $S_j.x$ such that the source for the reference is affected after the deletion of statement.

2. Find the values of $S_j.x$ for which the new source have to be recomputed.

3. Compute the new sources for the values found in (2).

First the references that are source for $S.1$ are found in statement 2-4. Then the references affected as a result of decrease in the generated values is computed in statement 5. The dependences for a reference $S_j.x$ is affected if one of its source is $S.1$. The values of $S_j.x$ that become uncovered are computed in statements 6-8. A value of $S_j.x$ becomes uncovered if its source is $S.1$. The variable `notCovered` contains the values of $S_j.x$ which are uncovered.

The new sources for the values in `notCovered` are computed in statements 9-11. The possible new sources for the values of $S_j.x$ in `notCovered` are the references in `source`. The new dependences are computed by the function *FindDep*. The dependences of $S_j.x$ not crossing statement S are not affected. Hence, the search for dependence is to be started from the statement preceding S.

Example : Consider the program in figure 6(b), the dependences in the program are given in equation 4. The process of updating the dependences by the algorithm *IncrAnalysisDelStmt* on deletion of the statement S_3 is given below.

The set `source` computed in the statements 2-4 is $\{S_1.1,\ S_2.1\ \}$. $S_4.1$ is the only reference which has $S_3.1$ as its source. The `notCovered` for this reference is $S_4.1[i,j] \mid 1 \leq i,j \leq 20$. The search for the new source functions begins from dependence covered by Entry, the last common loop between the statements S_4 and S_3. The dependences computed by the function FindDep are

$$S_2.1[i_1,j_1] \rightarrow S_4.1[i_2,j_2] \quad | \quad 1 \leq i_2,j_2 \leq 20 \ \wedge \ even(j_2) \ \wedge \ i_1 = i_2 \ \wedge \ 2j_1 = j_2$$

Algorithm IncrAnalysisDelStmt
Input : Statement S : The statement deleted
Output: The updated dependences

1 `source` = ϕ
2 For all references $S_i.1$ such that there is dependence $DepRel_{S_iS.1}$.
3 `source` = `source` $\cup$ S_i
4 Endfor
5 For all references $S_i.x$ such that $DepRel_{SS_i.x} \wedge S_i \neq S$
6 For all dependences such that $DepRel_{SS_i.x}$
7 `notCovered` = `notCovered` $\cup$ Range($DepRel_{SS_i.x}$)
8 Endfor
9 `beginLoop` = lastCommonLoop(S_i, S)
10 `beginStmt` = PrecStmt(S)
11 `newDeps` = FindDep($S_i.x$, `notCovered`, `beginStmt`, `source`)
12 Endfor

Figure 14: Incremental algorithm for deletion of an assignment statement

$$S_1.1[i_w, j_1] \rightarrow S_4.1[i_2, j_2] \quad | \quad 1 \leq i_2, j_2 \leq 20 \wedge odd(j_2) \wedge i_1 = i_2 \wedge j_1 = j_2$$

4.3 Insertion of loop

The insertion of a loop may cause both decrease and increase in the data generated as well as the values read as given in chapter 3. The algorithm **IncrAnalysisInsLoop**, given in figure 15 updates the dependences on insertion of a loop. In the algorithm the updating for increase and decrease in the generated values are combined together. The procedure of updating the dependences after insertion of a loop say, l is,

1. Find the source functions for all the references inside l.
2. Find the references outside l say $S_j.x$ such that the source function of $S_j.x$ are affected.
3. Find the values of $S_j.x$ that become uncovered after insertion of l. Compute values affected both for increase and decrease in the generated values.
4. Compute the souce functions for the values that become uncovered.

The variables used in the algorithm are

`oldSource` : Array of sets containing the old source references for the writes inside loop inserted.

`newSource` : Array of sets containing the new source references for the writes inside loop inserted.

`possibleSources` : The set of references that are possible source for the references affected.

The source functions for the references inside the loop l are computed in statements 2-8. The sets `oldSource` and `newSource` are are used for updating dependences as in the cases of increase/decrease in the generated values respectively.

There is change in the values generated by the writes inside only the loop l. The source for a reference outside l is affected as a result of decrease in the generated values if it has a source inside l. The new source for the values affected are the references in set `oldSource`. The source for a reference is affected as a result of increase in the generated values if its source is in set `newSource`. The new source for the reference can be the writes inside the loop l. A reference outside l, for which the dependence affected is to be updated for both increase and decrease in the values generated. Instead, the updates for both of these are combined together and updated in function *UpdateInsLoop*.

Let the references affected be $S.x$. The dependences affected for the reference $S.x$, and the values affected are computed in statement 13. A dependence $DepRel_{S,S.x}$ for $S.x$ is affected if it crosses the loop inserted l and the source is in set `source` or set `newSource`. This corresponds to the decrease and the increase of the values generated respectively. If a dependence $DepRel_{S,S.x}$ is affected, all the values of $S.x[i,s]$ in the dependence $DepRel_{S,S.x}$ become uncovered. The new sources for the values in `notCovered` are computed by the function FindDep. The possible sources for the values in `notCovered` are the references in set `possibleSources`. The variable `beginStmt` contains the statement from which the search for new dependence should start. The `beginStmt` is,

- the head of the last common loop between S and l, if S $\ll$ head of loop l.

Algorithm IncrAnalysisInsLoop
Input : Loop l : The loop inserted.
Output : The updated dependences after the insertion of the loop

1 `newSource` = `oldSources` = `possibleSources`= ϕ
2 For all references S.x inside the loop l
3 If reference S.x is a write
4 `oldSource`(S.x) = `oldSource`(S.x) $\cup$ (S_i such that $DepRel_{S,S.x}$)
5 Endif
6 $DepRel_{S.x}$ = FindDep($S.x, S, [S.x, s], U$)
7 If reference S.x is a write
8 `newSource`(S.x) = `newSource`(S.x) $\cup$ S_i such that $DepRel_{S,S.x} \epsilon DepRel_{S.x}$
9 `possibleSources` = `possibleSources` $\cup$ `newSource`(S.x) $\cup$ `oldSource`(S.x) $\cup$ S
10 Endif
11 Endfor
12 $\forall S_j.x$ outside l such that ($\exists DepRel_{S,S_j.x}$ $\wedge$ (S_i inside l $\vee$ $S_i \epsilon$ newSource))
13 UpdateInsLoop($S_j.x$, l, `possibleSources`)

Function UpdateInsLoop(reference S.x)
14 notCovered = ϕ
15 $\forall DepRel_{S,S.x}$ such that ($DepRel_{S,S.x}$ crosses l $\wedge$ (S_i inside l $\vee$ $S_i \epsilon$ newSource))
16 `notCovered` = `notCovered` $\cup$ Range($DepRel_{S,S.x}$)
17 l_n = lastCommonLoop(S.x, l)
18 If S.x $\ll$ l
19 `beginStmt` = For statement of l_n
20 else
21 `beginStmt` = Endfor of loop l_k such that l_k covers l and depth(l_k) = depth(l_n) + 1
22 Endif
23 FindDep(S.x, `beginStmt`, `notCovered`, `source`)

Figure 15: Incremental algorithm for insertion of do loop

- the Endfor of the loop l_k, where l_k covers l and if l_n is the last common loop between S and l, depth of l_k is depth(l_n) + 1.

It can be noted that in the exhaustive analysis algorithm, the dependences crossing l are computed after the search reaches the statement in `beginStmt`.

Example : Consider the program given in figure 9(a), the dependences in the program are given in equation 5. If a new loop is added as shown in figure 9(b), all the references to variable j in the statements S_4 and S_5 become references to the index j. The process of updating the dependences from the algorithm is given below.

The old dependences for the writes inside the loop is given in equation 5. The set oldSource is $\{S_1.1, S_2.1, S_3.1\}$. The new dependences for the writes inside the loop, $S_3.1$ and $S_4.1$ are,

$$S_1.1[i_1,j_1,k_1] \rightarrow S_4.1[i_2,j_2,k_2] \mid 1 \leq i_2 \leq 20 \wedge 2 \leq j_2 \leq 20 \wedge 1 \leq k_2 \leq 10 \wedge i_1 = i_2 \wedge j_1 = j_2 \wedge k_1 = 2k_2$$

$$S_2.1[i_1,j_1,k_1] \rightarrow S_4.1[i_2,j_2,k_2] \mid 1 \leq i_2 \leq 20 \wedge 2 \leq j_2 \leq 20 \wedge 11 \leq k_2 \leq 20 \wedge i_1 = i_2 \wedge j_1 = j_2 \wedge k_1 = k_2$$

$$S_1.1[i_1,j_1,k_1] \rightarrow S_3.1[i_2,j_2,k_2] \mid 1 \leq i_2,k_2 \leq 20 \wedge odd(k_2)\ 2 \leq j_2 \leq 20 \wedge i_1 = i_2 \wedge j_1 = j_2 \wedge k_1 = k_2$$

$$S_4.1[i_1,j_1,k_1] \rightarrow S_3.1[i_2,j_2,k_2] \mid 1 \leq i_2,k_2 \leq 20 \wedge even(k_2)\ 2 \leq j_2 \leq 20 \wedge i_1 = i_2 \wedge j_1 = j_2 \wedge 2k_1 = k_2$$

The set newSource is $\{S_3, S_2, S_1\}$ and the reference outside the loop l that is to be updated is only $S_5.1$.

The algorithm *UpdateInsLoop* calculates the new dependences for the reference $S_5.1$. The dependences affected for the reference are $DepRel_{S_1S_5.1}$ and $DepRel_{S_3S_5.1}$. The variable notCovered is $S_5.1[i,j,k] \mid 1 \leq i,j,k \leq 20$. The reference $S_5.1$ is lexically after the loop l. The last common loop between the S_4 and l_2 is Entry. The depth of Entry is taken as 0. The loop covering l_2 and at depth 1 is l_1. Hence, the search starts from Endfor statement of the loop l_1. The dependences for the reference is,

$$S_3.1[i_1,j_1,k_1] \rightarrow S_5.1[i_2,j_2,k_2] \mid 1 \leq i_2,k_2 \leq 20 \wedge 2 \leq j_2 \leq 20 \wedge i_1 = i_2 \wedge j_1 = j_2 \wedge k_1 = k_2$$

$$S_1.1[i_1,j_1,k_1] \rightarrow S_5.1[i_2,j_2,k_2] \mid 1 \leq i_2,k_2 \leq 20 \wedge j_2 = 1 \wedge i_1 = i_2 \wedge j_1 = j_2 \wedge k_1 = k_2$$

4.4 Deletion of loop

Deletion of a loop, say l, may cause both increase and decrease of the data generated. The incremental analysis on deletion of a loop is done by the algorithm

Algorithm IncrAnalysisDelLoop
Input : Loop l : The loop deleted.
Output : The updated dependences after the deletion of the loop

1 `newSource` = `oldSources` = `possibleSources`= ϕ
2 $\forall S.x$ such that S inside l and $x \neq 1$
FindDep($S.x, S, [S.x, s], U$)
3 For all references $S.x$ inside the loop l such that $x = 1$
4 `oldSource`(S.x) = `oldSource`(S.x) $\cup$ (S_i such that $DepRel_{S_i,S.x}$)
5 $DepRel_{S.x}$ = FindDep($S.x, S, [S.x, s], U$)
6 `newSource`(S.x) = `newSource`(S.x) $\cup$ S_i such that $DepRel_{S_i,S.x} \epsilon DepRel_{S.x}$
7 `possibleSources` = `possibleSources` $\cup$ `newSource`(S.x) $\cup$ `oldSource`(S.x) $\cup$ S
8 Endfor
9 $\forall S_j.x$ outside l such that ($\exists DepRel_{S_i,S_j.x} \wedge$ (S_i inside $l \vee S_i \epsilon$ newSource))
10 UpdateDelLoop($S_j.x$, l, `possibleSources`)

Algorithm UpdateDelLoop(reference S.x, loop l, set source)
11 `notCovered` = NULL
12 $\forall DepRel_{S_i,S.x}$ such that $DepRel_{S_i,S.x}$ l
`notCovered` = `notCovered` $\cup$ Range($DepRel_{S_i,S}$)
13 commonLoop = lastCommonLoop($S.x, l$)
14 If S.x $\ll$ l
15 `beginStmt` = commonLoop($S.x, l$)
16 Else
17 If (S and l are inside same loop)
18 `beginStmt` = Statement preceding Endfor of the loop l
19 Else
20 `beginStmt` = Endfor of loop l_k such that l_k covers l and depth(l_k) = depth(l_n) + 1
21 Endif
22 Endif
23 FindDep(S.x, `beginStmt`, `notCovered`, `source`)

Figure 16: Incremental algorithm for deletion of do loop

IncrAnalysisDelLoop given in figure 15. The dependences of a reference affected, are to be updated for both increase and decrease in the values generated. Instead, both the analysis are combined in the algorithm.

The functionalities of algorithms, *IncrAnalysisInsLoop* and *IncrAnalysisDelLoop* are similar. They differ only in the calculation of `beginStmt` in function UpdateDelLoop. The variable `beginStmt` is computed as,

- the head of the last common loop between S and l, if S $\ll$ head of loop l.
- the statement preceding Endfor of l if, both S and l are inside the same loop.
- the Endfor of the loop l_k, where l_k covers l and if l_n is the last common loop between S and l, depth of l_k is depth(l_n) + 1.

Example : Consider the program given in figure 9(b). If the loop l_2 is deleted, the program is as given in figure 9(a). The process of updating the dependences on deletion of l_2 is given below.

The dependences of the references inside the loop l_2 before the deletion of the loop are,

$$S_4.1[i_1,j_1,k_1] \rightarrow S_3.1[i_2,j_2,k_2] \mid 1 \leq i_2,k_2 \leq 20 \wedge even(k_2) \wedge 2 \leq j_2 \leq 20 \wedge i_1 = i_2 \wedge j_1 = j_2 \wedge 2k_1 = k_2$$

$$S_1.1[i_1,j_1,k_1] \rightarrow S_3.1[i_2,j_2,k_2] \mid 1 \leq i_2,k_2 \leq 20 \wedge odd(k_2)\ 2 \leq j_2 \leq 20 \wedge i_1 = i_2 \wedge j_1 = j_2 \wedge k_1 = k_2$$

$$S_1.1[i_1,j_1,k_1] \rightarrow S_4.1[i_2,j_2,k_2] \mid 1 \leq i_2 \leq 20 \wedge 2 \leq j_2 \leq 20 \wedge 1 \leq k_2 \leq 10 \wedge i_1 = i_2 \wedge j_1 = j_2 \wedge k_1 = 2k_2$$

$$S_2.1[i_1,j_1,k_1] \rightarrow S_4.1[i_2,j_2,k_2] \mid 1 \leq i_2 \leq 20 \wedge 2 \leq j_2 \leq 20 \wedge 11 \leq k_2 \leq 20 \wedge i_1 = i_2 \wedge j_1 = j_2 \wedge k_1 = k_2$$

The set `oldSource` = $\{S_1, S_2, S_3\}$. The new dependences for the references after the deletion of loop l are,

$$S_4.1[i_1,k_1] \rightarrow S_3.1[i_2,k_2] \mid 1 \leq i_2,k_2 \leq 20 \wedge even(k_2) \wedge i_1 = i_2 \wedge 2k_1 = k_2$$

$$S_1.1[i_1,j_1,k_1] \rightarrow S_3.1[i_2,k_2] \mid 1 \leq i_2,k_2 \leq 20 \wedge odd(k_2) \wedge i_1 = i_2 \wedge j_1 = 1$$

$$k_1 = k_2$$

$$S_1.1[i_1, j_1, k_1] \rightarrow S_4.1[i_2, k_2] \mid 1 \leq i_2 \leq 20 \ \wedge \ 1 \leq k_2 \leq 10 \ \wedge \ j_1 = 1 \ \wedge \ i_1 = i_2 \ \wedge \ k_1 = 2k_2$$

$$S_2.1[i_1, j_1, k_1] \rightarrow S_4.1[i_2, k_2] \mid 1 \leq i_2 \leq 20 \ \wedge \ j_1 = 1 \ \wedge \ 11 \leq k_2 \leq 20 \ \wedge \ i_1 = i_2 \ \wedge \ k_1 = k_2$$

The set newSource = $\{S_1, S_2, S_3\}$. $S_5.1$ is the only reference outside the loop l that has its dependence affected. Its dependences are updated by the algorithm *UpdateDelLoop*. The `lastStatement` for the deleted loop l_2 is the Endfor statement of the loop l_3. The `notCovered` for the reference S_5 is $S_5.1[i, j, k] \mid 1 \leq i, j, k \leq 20$. The start statement is the Endfor statement of the last common loop between S_5 and l_2, i.e Endfor of loop l_1. The new source functions for the reference $S_5.1$, calculated by the function *FindDep* are,

$$S_3.1[i_1, k_1] \rightarrow S_5.1[i_2, j_2, k_2] \mid 1 \leq i_2, k_2 \leq 20 \ \wedge \ j_2 = 1 \ \wedge \ i_1 = i_2 \ \wedge \ k_1 = k_2$$

$$S_1.1[i_1, j_1, k_1] \rightarrow S_5.1[i_2, j_2, k_2] \mid 1 \leq i_2, k_2 \leq 20 \ \wedge \ 2 \leq j_2 \leq 20 \ \wedge \ i_1 = i_2 \ \wedge \ j_1 = j_2 \ \wedge \ k_1 = k_2$$

Chapter 5

Results

In this chapter we will present the timings for incremental analysis done by using the algorithms given in chapter 4, as compared to the timings for exhaustive analysis.

The timings are taken for 6 programs, each of about 50 lines from NASA NAS benchmark programs provided with the tiny tool. In the graphs given below, the fraction of incremental analysis time over exhaustive analysis time is given in the X-axis. The fractions are rounded to nearest multiple of 0.05. Y-axis contains the percentage of cases that gave the speedup. We present some of the interesting observations corresponding to the speedups obtained.

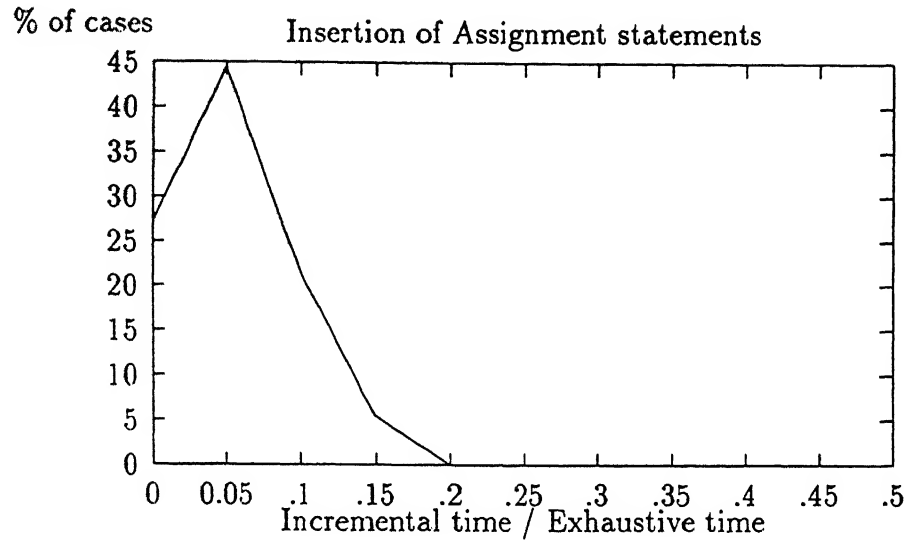
5.1 Insertion of assignment statmenent

The time for incremental analysis on insertion of an assignment statement is 0.02 to 0.2 times that of exhaustive analysis time, as given in the graph below.

Insertion of an assignment statement, say S affects the dependences corresponding to the variable Arr(S.1). Dependences involving other variables are not affected. In programs that use many variables, the incremental analysis, on average, takes much lesser time than the time for exhaustive analysis.

Consider a statement is inserted such that, it overwrites values written by E. That is, it is first write to some memory locations in the program. The possible source for the affected references are the write inserted and the Entry only. Hence,

the search space is vary small and the speedup is more.

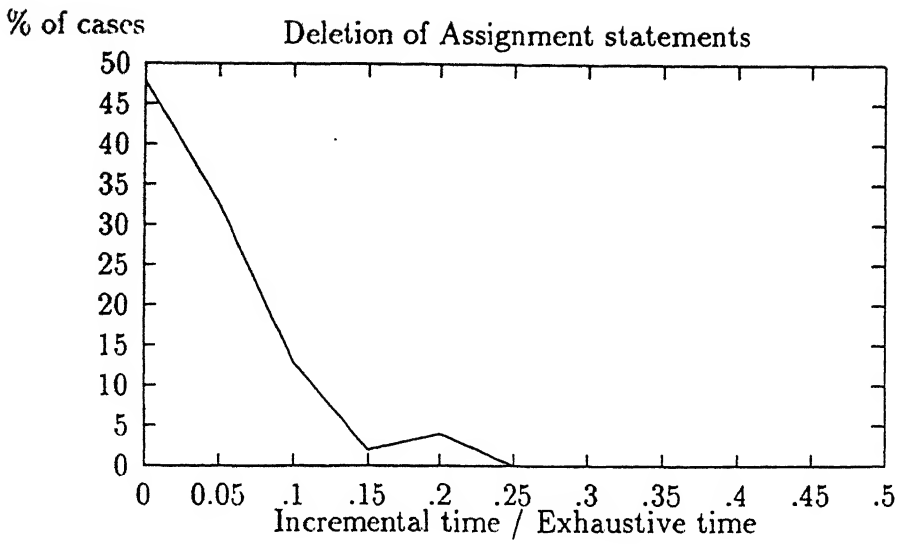


5.2 Deletion of assignment statement

The time for incremental analysis on deletion of an assignment statement is 0.02 to 0.25 times that of exhaustive analysis as given in the graph below.

Deletion of an assignment statement S affects the dependences corresponding to the variable Arr(S.1). In programs that use many variables, the incremental analysis, on average, takes much lesser time than the time for exhaustive analysis.

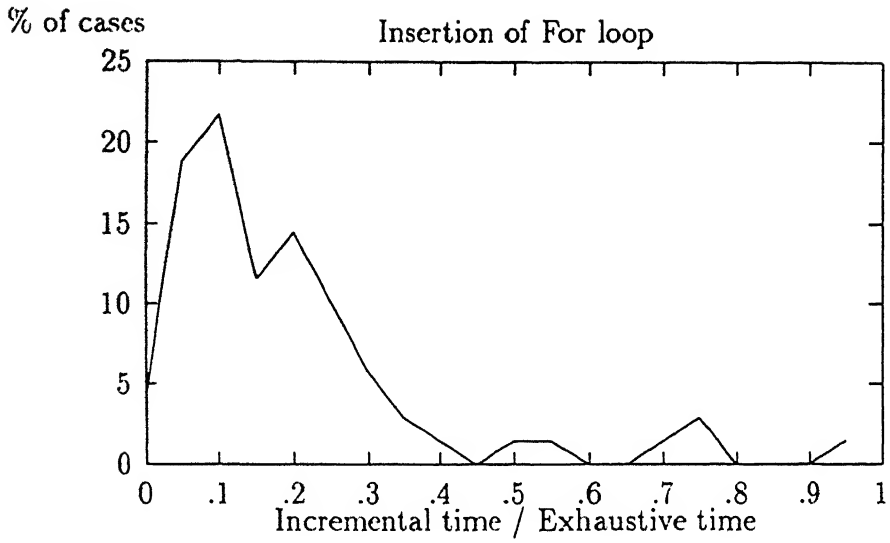
If a write having less number of source references is deleted, the search space for the sink references is very small. In such cases, the incremental analysis is expected to give significant speedups. In particular if the source of the deleted write is only the Entry, the speedup is very significant as no dependence testing is needed.

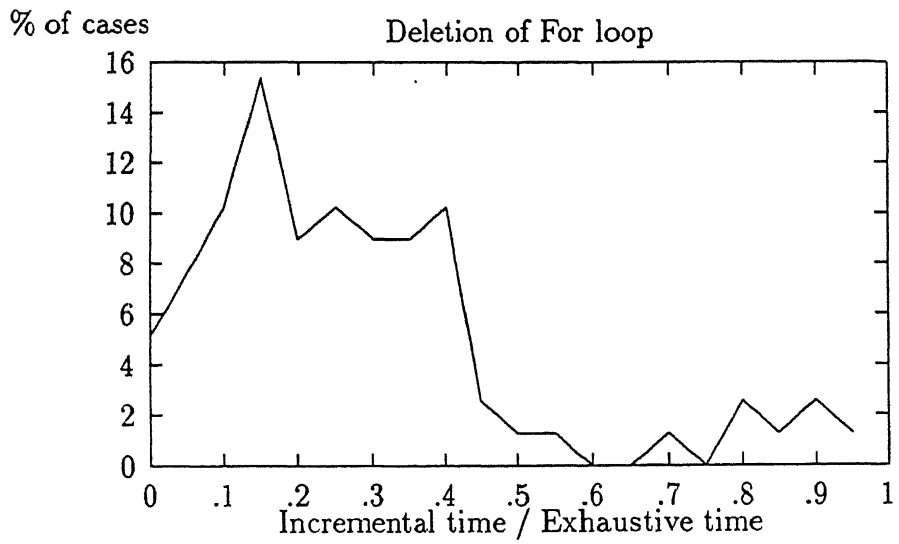


5.3 Insertion/Deletion of loop

The time for incremental analysis on insertion of an assignment statement is 0.05 to 0.95 times the exhaustive analysis time. For majority of the cases the incremental analysis time is 0.1 to 0.25 times the exhaustive analysis time.

In most of the cases, the percentage of assignment statements inside the loop compared to those in the program is 10 to 30%. In such cases, the time for incremental analysis is 0.1 to 0.4 times the exhaustive analysis time. In cases where the percentage of assignment statements in the loop is more than 40%, the time for incremental analysis also increases. If percentage of assignment statements inside the loop is more than about 80%, the time for incremental analysis is almost equal to exhaustive analysis. This is because insertion/deletion of such loops affects almost all the references inside the program. Hence, all the dependences have to be recomputed.





Chapter 6

Conclusion

In this work, the incremental dependence analysis for value based dependence has been presented. We have shown how the dependences change as a result of arbitrary increase or the decrease in the values written or read in a program.

We have shown how incremental analysis for addition and deletion of assignment statements, and for loops can be formulated as increase/decrease in the values generated. The algorithms for insertion and deletion of assignment statements and loops have been presented. The approach for incremental analysis is general enough for other form modifications in a straight-forward way.

6.1 Future Directions

- The algorithms developed are for single modifications. An interesting problem to investigate would be to find how the dependences are affected as a result of multiple modifications. We claim that updates in these cases can be done similar to insertion/deletion of loops and we have addressed most of the issues.
- We have given algorithms for insertion/deletion of assignment statements and loops. This can be extended to other type of modifications also.
- The work address incrementalism in analysis procedure. An orthogonal view would be to incrementalize the dependence testing. This is an interesting problem to investigate.

- Our implementation is over the UMCP version of *Tiny tool.* The tool was not developed for incremental analysis and hence many overheads are present. A complete implementation with incremental analysis as goal may produce better results in some cases.

- The semantics defined on the index variable of the loop plays an important role in values generated after the insertion/deletion of a loop. For example, if the index variable is i, after the execution of the loop the value of i may be,

  the value of i in the last iteration of the loop or

  an undefined value or

  any value that the index i took. A comparative study of the performances is an important issue in practical cases.

Bibliography

[1] Dror E Maydan Saman P. Amarasinghe and Monica S Lam. Array data-flow analysis and its use in array privatization. In *ACM'93 Conference on Principles of Programming Languages*, January 1993.

[2] Paul Feautrier. Array expansion. In *ACM International Conference On Supercomputing*, pages 429 441, 1988.

[3] Paul Feautrier. Data flow analysis of array and scalar references. *Journal of Parallel Programming*, 20(1), Febrauary 1991.

[4] Bill Applebe Kevin Smith and Kurt Stirewalt. Incremental dependence analysis for interactive parallelization. *Proceedings 1990 ACM International Conference on Supercomputing*, pages 330 341, June 1990.

[5] B. Kuck D.J. Kuhn, R.H. Leasure and Wolfe M. The structure of an advanced vectorizer for pipelined processors. In *Proceedings of COMPSAC 80, the* 4^{th} *International Computer Software and Applications Conference*, October 1980.

[6] Wolfe M. *Optimizing Supercompilers for Supercomputers.* MIT Press, 1989.

[7] Vadim Maslov. Array data-flow dependence analysis. In *ACM'94 International conference on Principles of programming Languages*, pages 311–325, March 1994.

[8] Dror E Maydan, John L Hennessy, and Monica S Lam. Effectiveness of data dependence analysis. *Journal of Parallel Programming*, pages 330–341, 1995.

[9] C.D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Press, 1988.

[10] William Pugh. Uniform techniques for loop optimization. In *1991 International conference on Supercomputing*, pages 341–352, June 1991.

[11] William Pugh and David Wonnacott. An evaluation of exact methods for analysis of value based dependence array data dependences. In *Sixth Annual Workshop on Programming Languages and Compilers for Parallel Computing*, August 1993.

Appendix A

Exhaustive Value Based Dependence Analysis

A.1 Lazy Array Data-Flow Dependence Analysis

The value based dependence for a reference $S_1.k$, such that $S_1.k$ is sink is defined symbolically as,

$\forall r, s : (S_i[v, s] \rightarrow S_1.k[r, s]) \epsilon DepRel_{S_i S_1.k} \Leftrightarrow$

$max_{\ll}(S[w, s] \mid w \epsilon [S, s] \wedge r \epsilon [S_1.k, s] \wedge Arr(S.1) = Arr(S_1.k) \wedge S[w, s] \ll S_1.k[r, s])$

Vadim Maslov [7] proposed a procedure to compute source functions, $DepRel_{S_i.k}$, for a reference $S_i.k$ based on the definition given above definition. The procedure is called "Lazy Array Data-Flow Analysis ". The procedure can be considered as a search for the most recent hits to the memory locations referred to by values of $S_i.k$. It takes advantage of the fact that the most recent hit is the first one encountered while moving backward in the iteration space. If l_n is the innermost loop covering statement $S_i.k$, first the loop independent dependences carried by loop l_n are computed. If source for some values of $S_i.k$ are not yet computed, the dependences carried by l_n are computed. Then the process is repeated for next higher loop say l_{n-1} and so on. This process is repeated till the source for all the values of $S_j.k$ are computed or all the possible writes are considered. If for some values there are no sources, they are made to be dependent on Entry.

The algorithm **FindDep** given in figure 17 is a modified form of the algorithm given in [7]. The original algorithm is modified to accept different inputs.

The variables used in the algorithm are,

$S_j.x$: The reference for which the source functions are to be computed.

`notCovered` : Contains the values of reference $S_j.x$ for which the source are to be computed. For exhaustive analysis, `notCovered` = $[S_j.x, s]$.

`startStmt` : Contains the statement from which the search for dependence is to be started. For exhaustive analysis `startStmt` = S_j.

`source` : A set consisting of the references which can be source for $S_j.x$. For exhaustive analysis `source` = U, the universal set.

`DepRel` : Contains the dependence relations computed in the algorithm.

`cMax` : When there are more then one possible writes, this contains the dependence from a single source.

`possibleSources` : The set of references that can be sources for values in `notCovered`

The search for the sources start from the statement `beginStmt`, and `depLoop` is set to the loop covering statement R. The search moves backwards lexically. If the statement reached is a

an *assignment* **statement** S_i: The dependence from S_i to $S_1.k$ is computed (as in statement 11).

a *do* **statement** l_n : The search is crossing the loop boundary, a values generated in the previous iterations of the loop l_n may be the source. Hence, the loop carried dependences from the writes inside l are computed as in statement 20 by the function *FindCoverMultipleWrite*.

an *Enddo* **of loop** l_n: The search is entering into a loop l_n, hence the loop independent dependences from all statements inside l_n are to be computed as in statement 20.

Entry There is no write to the memory locations of values in **notCovered**. Hence, they are made dependent on Entry.

Example: Consider the program given in figure 18. The process of finding the dependence for the reference $S_1.1$ i.e, write $XRISQ(i,q)$ in statement S_1, by the algorithm is given below.

The variable **notCovered** contains all the values generated by the reference. The values in **notCovered** are,

$$S_1.1(rs,p,q,i) \mid 1 \leq rs \leq nrs \wedge 1 \leq p \leq np \wedge 1 \leq q \leq p \wedge 1 \leq i \leq mb.$$

This is an exhaustive analysis hence, the variable **beginStmt** $= S_1$. First the loop independent dependences carried by the loop l_3 are computed. There is no write inside the loop l_3 preceding statement S_1 and hence no loop independent dependence carried by l_3.

The statement preceding S_1 is the *Do* statement of the loop l_3. The search is crossing loop l_3 and hence the loop carried dependences carried by l_3, are to be computed. The flag **carriedFlag** is set to TRUE. The possible writes are S_2 and S_1. The dependences from these writes is computed by the function *FindCoverSingleWrite* in statement 11. There is no loop carried dependence carried by l_3 from $S_2.1$ to $S_1.1$ and from $S_1.1$ to $S_2.1$.

The statement preceding *Do* of l_3 is *Do* of loop l_2. Hence, the dependences carried by l_2 are computed in statements 15-20. The possible sources are S_2 and S_1. There is no dependence from $S_1.1$ to $S_1.1$ carried by loop l_2. The dependence from $S_2.1$ to $S_1.1$ carried by loop l_2 is,

$$S_2[p_1,q_1,i_1] \rightarrow S_1[p_2,q_2,i_2] \mid 2 \leq p_2 = q_2 \leq np \wedge 1 \leq i_2 \leq mb \wedge p_1 = p_2 \wedge q_1 = q_2 - 1 \wedge i_1 = i_2$$

The values in of $S_1.1$ in Range($DepRel_{S_2S_1}$) is removed from **notCovered** in statement 26. The resulting **notCovered** is

$$S_1.1[p,q,i] \mid (p = q = 1 \wedge 1 \leq i \leq mb) \vee (1 \leq q < p \leq np \wedge 1 \leq i \leq mb)$$

```
Algorithm FindDep
INPUT :
    Reference S_j.x : The reference for which the dependence is to be found
    such that its the sink.
    DNF notCovered : The values for which the dependences are to be found
    Statement startStmt : The statement from which the search starts
    Set source : The set of statements which are possible sources
OUTPUT :
    Dependence Relations DepRel_{S_j.x}
1   Relation DepRel = φ
2   Statement loop = CoveringLoop( S_j.x )
4   Boolean carriedFlag = FALSE
5   Statement S = startStmt
6   While (notCovered is feasible) do
7       S = statement preceding S
8       wrMax = φ
9       Switch (type of statement S)
10      case assignment statement
11          wrMax = FindCoverSingleWrite(S_j.x, {S.1}, notCovered, lastCommonLoop(S, S_j), FALS
12          break;
13      case Do of loop l_n
14      case Enddo of loop l_n
15          If (S is Do statement)
16              carriedFlag = TRUE
17          Else
18              carriedFlag = FALSE
19          possibleSources = {S_i.1 | S_i inside l_n and S_i ϵ source}
20          wrMax = FindCoverMultipleWrite(S_j.x, possibleSources, notCovered, l_n, carriedFlag)
21          break;
22      case Entry
23          wrMax = DepRel ∪ { Entry → S_j.x[i_1, s]' | S_j.x[i_1, s] ϵ notCovered }
24          break
25      Endswitch
26      notCovered = notCovered − Range(wrMax)
27      DepRel = DepRel ∪ wrMax
28  Endwhile
29  return DepRel

Function FindCoverMultipleWrite(Reference S_j.x, set possibleSources, DNF notCovered,
                                 Statement loop, Flag carriedFlag)
30  cMax = φ
31  Forall writes S_i.1 such that S_i ϵ possibleSources
32      cMax = FindCoverSingleWrite(S_j.x, S_i.1, notCovered, l_n, carriedFlag)
33      wrMax = RelMax_2(wrMax, cMax)
33  Endfor
34  return wrMax
```

Figure 17: Algorithm for Exhaustive Dependence Analysis

```
l1          Do q = 1, np
              Do i = 1, mb
S0              XRSIQ(i,q) = 0
              Enddo
            Enddo
l1          Do p = 1, np
l2            Do q = 1, p
l3            Do i = 1, mb
S1              XRSIQ(i,q) = XRSIQ(i,q)+ ...
S2              XRSIQ(i,p) = XRSIQ(i,p)+ ...
              Enddo
            Enddo
```

Figure 18: Code from subroutine OLDA from TRFD

Then the dependences carried by loop l_1 is found. The dependences from statements S_2 and S_1 are,

$$S_2[p_1, q_1, i_1] \rightarrow S_1[p_2, q_2, i_2] \mid 1 \leq q_2 < p_2 \leq np \ \wedge \ 1 \leq i_2 \leq mb \ \wedge \ p_1 = q_1 = q_2 \ \wedge \ i_1 = i_2$$

$$S_1[p_1, q_1, i_1] \rightarrow S_1[p_2, q_2, i_2] \mid 1 \leq q_2 \leq p_2 \leq np \ \wedge \ 1 \leq i_2 \leq mb \ \wedge \ p_1 = p_2 - 1 \ \wedge \ q_1 = q_2 \ \wedge \ i_1 = i_2$$

The lexicographical maximum of these two candidate writes is found by $RelMax_2$ in statement 21. The resulting dependences are,

$$S_2[p_1, q_1, i_1] \rightarrow S_1[p_2, q_2, i_2] \mid 2 \leq p_2 \leq np \wedge q_2 = p_2 - 1 \wedge 1 \leq i_2 \leq mb \ \wedge \ p_1 = p_2 - 1 \ \wedge \ p_1 = p_2 \ \wedge \ i_1 = i_2$$

$$S_1[p_1, q_1, i_1] \rightarrow S_1[p_2, q_2, i_2] \mid p_2 \leq np \wedge 1 \leq q_2 \leq p_2 - 2 \wedge 1 \leq i_2 \leq mb \ p_1 = p_2 - 1 \ \wedge \ q_1 = q_2 \ \wedge \ i_1 = i_2$$

The values covered by these dependences are removed from the notCovered, the resulting notCovered is,

$$S_1[p, q, i] \mid (p = q = 1 \wedge 1 \leq i \leq mb).$$

The search moves backwards, and reaches the *Enddo* of the loop l_1. The only write inside the loop l_1 is S_0, the dependence covered by l_1 from S_0 to S_1 is computed. This is the only candidate write, and the dependences for the values in notCovered are,

$$S_1[q_1, i_1] \rightarrow S_1[p_2, q_2, i_2] \mid p_2 = q_2 = 1 \wedge 1 \leq i_2 \leq mb \wedge q_1 = p_2 \wedge i_1 = i_2$$

The values covered in this dependence are removed from the notCovered. The notCovered results to ϕ and the search stops.